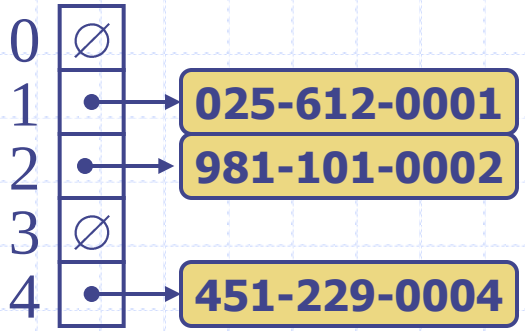
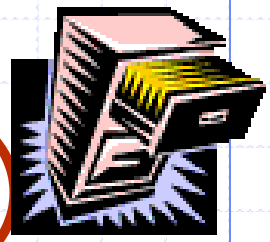
Tabelle hash

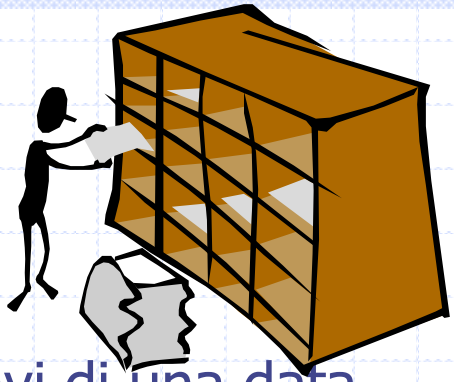


Richiamo TDA mappa (9.1)



- Metodi del TDA mappa:
  - get(k): se la mappa M ha una entry con chiave k, restituisce il valore ad essa associato, altrimenti restituisce null
  - put(k, v): inserisce nella mappa M la entry (k, v); se la chiave k non è già in M restituisce il risultato null; altrimenti, se w è il valore già presente associato a k, sostituisce w con v, restituendo come risultato proprio w
  - remove(k): se la mappa M ha una entry con chiave k, rimuove tale entry da M e restituisce il valore associato a k; altrimenti, restituisce null
  - size(), isEmpty()
  - keys(): restituisce una collection iterabile contenente tutte le chiavi contenute in M (keys().iterator() restituisce un iteratore alle chiavi)
  - values(): restituisce una collection iterabile contenente tutti i valori associati alle chiavi contenute in M (values().iterator() restituisce un iteratore ai valori)
  - entries(): restituisce una collection iterabile contenente tutte le entry chiave-valore contenute in M (entries().iterator() restituisce un iteratore alle entry)

Funzioni hash e tabelle hash (9.2)

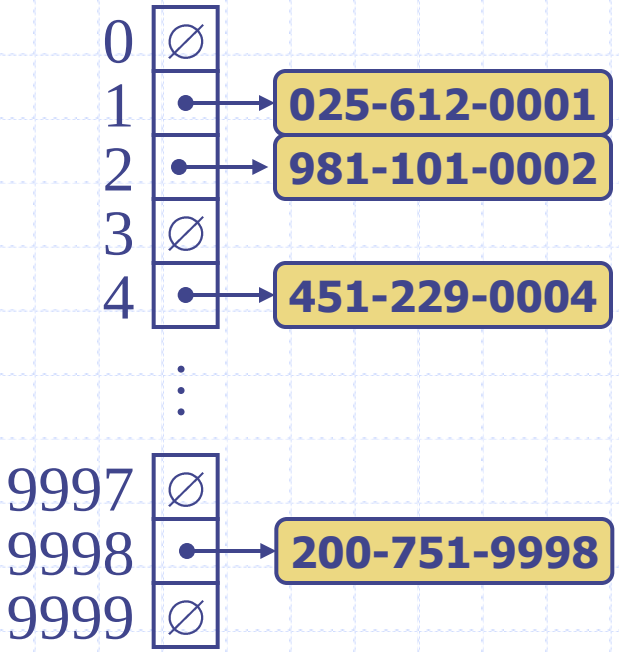


- Una funzione hash h trasforma (mappa) chiavi di una data tipologia in interi appartenenti a un intervallo prefissato $[0, N-1]$
- Esempio:
  $$h(x) = x \bmod N$$
  è una funzione hash per chiavi intere
- L'intero $h(x)$ è detto codice hash della chiave x

- Una tabella hash per una data tipologia di chiavi consiste di
  - una funzione hash h
  - un array (tabella) di dimensione N
- Nell'implementazione di una mappa attraverso una tabella hash, l'obiettivo è memorizzare l'elemento (k, o) nella posizione $i = h(k)$

Esempio

- Possibile tabella hash per una mappa che memorizza entry del tipo (SSN, Nome), dove SSN (social security number) è un intero positivo di nove cifre
- La Tabella hash usa un array di dimensione $N = 10000$ e la funzione hash $h(x) =$ ultime quattro cifre di x



Funzioni Hash (§ 9.2.2)

- Una funzione hash è di solito specificata come composizione di due funzioni:

  Codice hash:

  h_1: chiavi $\rightarrow$ interi

  Funzione di compressione:

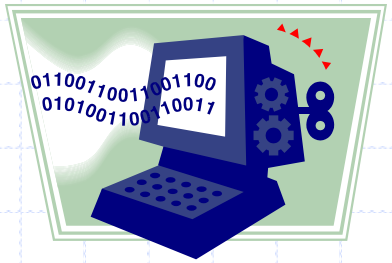
  h_2: interi $\rightarrow [0, N-1]$

- Si applica dapprima il codice hash; successivamente la funzione di compressione viene applicata sul risultato, cioè

  $$h(x) = h_2(h_1(x))$$

- L'obiettivo della funzione hash è "disperdere" le chiavi in maniera apparentemente casuale

Codici Hash (9.2.3)



- Codici hash in Java:
  - Si interpreta l'indirizzo di memorizzazione dell'oggetto chiave come intero (metodo java.lang.Object.hashCode(), che restituisce un int)
  - Buono in generale, eccetto che per chiavi numeriche e stringhe
- Assegnazione a un numero intero:
  - Si reinterpretano i bit della chiave come intero
  - Tecnica idonea per chiavi di lunghezza non superiore al numero di bit del tipo intero (in Java: byte, short, int, float)
- Somma di componenti:
  - Si partizionano i bit della chiave in componenti di lunghezza prefissata (ad es., 16 o 32 bit) e si sommano tali componenti (ignorando gli overflow)
  - Idonea per chiavi numeriche di lunghezza almeno uguale al numero di bit del tipo intero (in Java: long e double)

Codici hash (cont.)

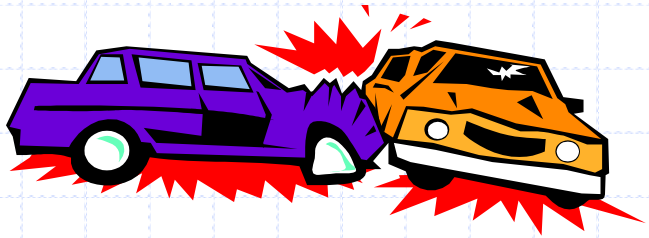
- Codici hash polinomiali:
  - Si partizionano i bit della chiave in una sequenza di componenti di lunghezza fissata (ad es., 8, 16 o 32 bit)
    $$a_0 a_1 \ldots a_{n-1}$$
  - Si valuta il polinomio
    $$p(z) = a_0 + a_1 z + a_2 z^2 + \ldots + \ldots + a_{n-1} z^{n-1}$$
    su un valore z prefissato, ignorando gli eventuali overflow
  - Particolarmente idoneo su stringhe (ad es., la scelta $z = 33$ determina al più 6 collisioni in un insieme di 50000 vocaboli della lingua inglese)

- Il polinomio $p(z)$ può essere valutato in tempo $O(n)$ usando la regola di Horner:
  - calcolare in sequenza i seguenti polinomi, ciascuno ottenuto dal precedente in tempo $O(1)$
    $$p_0(z) = a_{n-1}$$
    $$p_i(z) = a_{n-i-1} + z p_{i-1}(z) \quad (i = 1, 2, \ldots, n-1)$$
  - Vale $p(z) = p_{n-1}(z)$

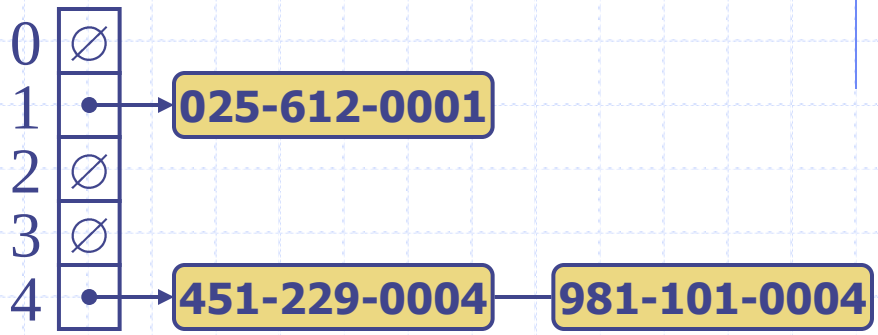
Funzioni di compressione (9.2.4)

- Il metodo di divisione:
  - $h_2(y) = y \bmod N$
  - La dimensione N della tabella hash viene di solito scelta come numero primo
  - Se N non è primo, eventuali pattern nella distribuzione di codici hash possono ripetersi nella distribuzione dei valori hash, causando collisioni
- Il metodo MAD (moltiplicazione, addizione e divisione):
  - $h_2(y) = |ay + b| \bmod N$
  - a e b sono interi casuali tali che
    - $a > 0$
    - $a \bmod N \neq 0$
    - $b \geq 0$
  - Se $a \bmod N = 0$ ogni intero verrebbe trasformato nello stesso valore b

Gestione delle collisioni (§ 9.2.5)



- Si verificano collisioni quando elementi differenti vengono mappati nella stessa cella
- **Concatenazione separata**: ad ogni cella nella tabella viene associata una lista collegata di entry che vengono mappate in quella stessa cella



- La concatenazione separata è semplice ma richiede spazio di memoria addizionale all'esterno della tabella

Metodi della mappa con gestione collisioni a concatenazione separata

- Tabella hash con concatenazione separata: metodi fondamentali.

Algorithm get(k):
Output: Il valore associato alla chiave k della mappa, o **null** se non esiste nessuna entry con chiave k
return $A[h(k)]$.get(k) {delega il get alla mappa basata su lista a $A[h(k)]$}

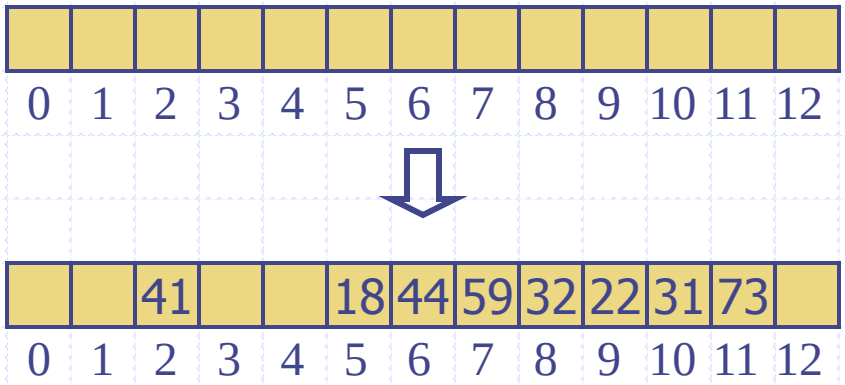
Algorithm put(k,v):
Output: Se esiste una entry con chiave k, restituisce il suo valore (che viene sostituito da v); altrimenti restituisce **null**
$t = A[h(k)]$.put(k,v) {delega il put alla mappa basata su lista a $A[h(k)]$}
if $t =$ **null then** {k è una chiave nuova}
    $n = n + 1$
return t

Algorithm remove(k):
Output: Il valore (cancellato) associato alla chiave k, o **null** se non esiste la chiave k nella mappa
$t = A[h(k)]$.remove(k) {delega il remove alla mappa basata su lista a $A[h(k)]$}
if $t \neq$ **null then** {k trovata}
    $n = n - 1$
return t

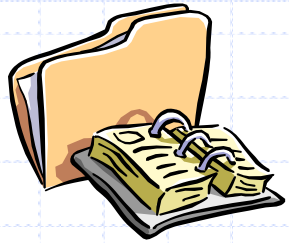
Linear probing

- Indirizzamento aperto: l'elemento collidente viene inserito in un'altra cella della tabella
- **Linear probing** gestisce le collisioni piazzando l'elemento collidente nella prossima (in senso circolare) cella disponibile della tabella
- Ciascuna ispezione di cella viene denominata "probe"
- Elementi collidenti tendono ad ammassarsi, causando future sequenze di probe più lunghe

- Esempio:
  - $h(x) = x \bmod 13$
  - Inserimento ordinato delle chiavi 18, 41, 22, 44, 59, 32, 31, 73



Ricerca con linear probing



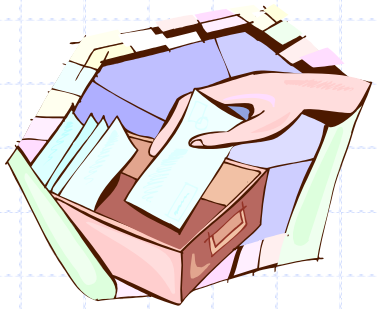
- Consideriamo una tabella hash A che usa linear probing
- get(k)
  - si inizia dalla cella $h(k)$
  - si sondano locazioni consecutive finché non accade uno dei seguenti eventi
    - viene trovata una entry con chiave k, oppure
    - viene trovata una cella vuota, oppure
    - sono state infruttuosamente sondate N celle

```
Algorithm get(k)
    i ← h(k)
    p ← 0
    repeat
        c ← A[i]
        if c = ∅
            return null
        else if c.key () = k
            return c.element()
        else
            i ← (i + 1) mod N
            p ← p + 1
    until p = N
    return null
```

Aggiornamenti con il linear probing

- Per gestire inserimenti e cancellazioni, introduciamo un oggetto speciale (sentinella) chiamato AVAILABLE che sostituisce gli elementi eliminati
- remove(k)
  - si cerca una entry con chiave k
  - se viene trovata una entry (k, o), questa viene sostituita con la sentinella e viene restituito l'elemento o
  - Altrimenti, si restituisce *null*
- put(k, o)
  - si lancia eccezione se la tabella è piena
  - si inizia a cella $h(k)$
  - si sondano consecutivamente le celle finché non viene trovata una cella i vuota o con l'oggetto sentinella
    - ciò accadrà sicuramente altrimenti sarebbe stata lanciata un'eccezione
  - memorizziamo la entry (k, o) nella cella i

Hashing doppio



- Lo hashing doppio usa una funzione di hash secondaria $d(k)$ e gestisce le collisioni posizionando gli elementi collidenti nella cella i nella prima cella disponibile secondo lo schema

  $$(i + j\,d(k)) \bmod N$$

  per $j = 0,\ 1, \ldots, N-1$
- La funzione $d(k)$ non può avere valori nulli
- La dimensione N della tabella deve essere un numero primo per garantire il probe di tutte le sue celle
- Una scelta comune della funzione di compressione per la funzione hash secondaria è:

  $$d_2(k) = q - (k \bmod q)$$

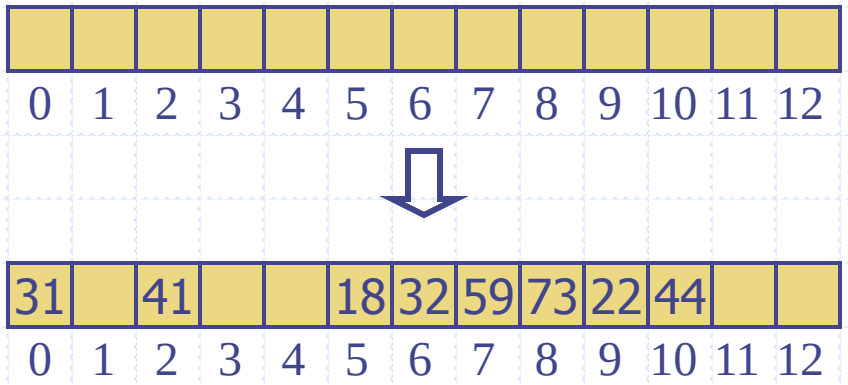
  dove
  - $q < N$
  - q è un numero primo
- I valori possibili di $d_2(k)$ sono

  $$1, 2, \ldots, q$$

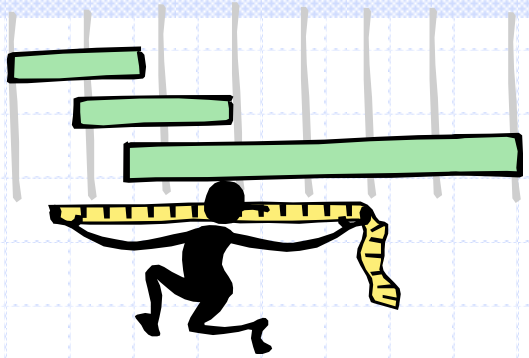
Esempio di hashing doppio

- Si consideri una tabella hash a chiavi intere che gestisce le collisioni con hashing doppio
  - $N = 13$
  - $h(k) = k \bmod 13$
  - $d(k) = 7 - k \bmod 7$
- Inserimento ordinato delle chiavi 18, 41, 22, 44, 59, 32, 31, 73

k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		



Le prestazioni dello hashing



- Nel caso peggiore, ricerche, inserimenti ed eliminazioni richiedono tempo $O(n)$
- Il caso peggiore si verifica quando tutti gli inserimenti causano collisione
- Le prestazioni in pratica risentono del fattore di carico $\alpha = n/N$
- Assumendo che i valori hash abbiano una distribuzione uniforme, si può dimostrare che il numero atteso di probe per inserimento, utilizzando l'indirizzamento aperto, è

$$1 / (1 - \alpha)$$

- Il tempo atteso di ogni operazione in una tabella hash è $O(1)$
- In pratica, lo hashing è estremamente veloce purché il fattore di carico non sia troppo vicino al 100%
- Applicazioni delle tabelle hash:
  - piccoli database
  - compilatori
  - cache dei browser

Java Example

```
/** A hash table with linear probing and the MAD hash function */
public class HashTable implements Map {
  protected static class HashEntry implements Entry {
    Object key, value;
    HashEntry () { /* default constructor */ }
    HashEntry(Object k, Object v) { key = k; value = v; }
    public Object key() { return key; }
    public Object value() { return value; }
    protected Object setValue(Object v) {  // set a new value, returning old
      Object temp = value;
      value = v;
      return temp;  // return old value
    }
  }
  /** Nested class for a default equality tester */
  protected static class DefaultEqualityTester implements EqualityTester {
    DefaultEqualityTester() { /* default constructor */ }
    /** Returns whether the two objects are equal.  */
    public boolean isEqualTo(Object a, Object b) { return a.equals(b); }
  }
  protected static Entry AVAILABLE = new HashEntry(null, null); // empty
        marker
  protected int n = 0;                  // number of entries in the dictionary
  protected int N;                      // capacity of the bucket array
  protected Entry[] A;                                  // bucket array
  protected EqualityTester T;           // the equality tester
  protected int scale, shift;   // the shift and scaling factors
  /** Creates a hash table with initial capacity 1023. */
  public HashTable() {
    N = 1023; // default capacity
    A = new Entry[N];
    T = new DefaultEqualityTester(); // use the default equality tester
    java.util.Random rand = new java.util.Random();
    scale = rand.nextInt(N-1) + 1;
    shift = rand.nextInt(N);
  }
```

```
/** Creates a hash table with the given capacity and equality tester. */
  public HashTable(int bN, EqualityTester tester) {
    N = bN;
    A = new Entry[N];
    T = tester;
    java.util.Random rand = new java.util.Random();
    scale = rand.nextInt(N-1) + 1;
    shift = rand.nextInt(N);
  }
```

Java Example (cont.)

```
/** Determines whether a key is valid. */
protected void checkKey(Object k) {
  if (k == null) throw new InvalidKeyException("Invalid key: null.");
}
/** Hash function applying MAD method to default hash code. */
public int hashValue(Object key) {
  return Math.abs(key.hashCode()*scale + shift) % N;
}
/** Returns the number of entries in the hash table. */
public int size() { return n; }
/** Returns whether or not the table is empty. */
public boolean isEmpty() { return (n == 0); }
/** Helper search method - returns index of found key or -index-1,
 * where index is the index of an empty or available slot. */
protected int findEntry(Object key) throws InvalidKeyException {
  int avail = 0;
  checkKey(key);
  int i = hashValue(key);
  int j = i;
  do {
    if (A[i] == null)   return -i - 1;  // entry is not found
    if (A[i] == AVAILABLE) {              // bucket is deactivated
        avail = i;                        // remember that this slot is available
        i = (i + 1) % N;                  // keep looking
    }
    else if (T.isEqualTo(key,A[i].key()))  // we have found our entry
        return i;
    else // this slot is occupied--we must keep looking
        i = (i + 1) % N;
  } while (i != j);
  return -avail - 1;  // entry is not found
}
/** Returns the value associated with a key. */
public Object get (Object key) throws InvalidKeyException {
  int i = findEntry(key);  // helper method for finding a key
  if (i < 0) return null;  // there is no value for this key
  return A[i].value();     // return the found value in this case
}
```

```
/** Put a key-value pair in the map, replacing previous one if it exists. */
public Object put (Object key, Object value) throws InvalidKeyException {
  if (n >= N/2) rehash(); // rehash to keep the load factor <= 0.5
  int i = findEntry(key); //find the appropriate spot for this entry
  if (i < 0) {       // this key does not already have a value
    A[-i-1] = new HashEntry(key, value); // convert to the proper index
    n++;
    return null;    // there was no previous value
  }
  else                                  // this key has a previous value
    return ((HashEntry) A[i]).setValue(value); // set new value & return old
}
/** Doubles the size of the hash table and rehashes all the entries. */
protected void rehash() {
  N = 2*N;
  Entry[] B = A;
  A = new Entry[N]; // allocate a new version of A twice as big as before
  java.util.Random rand = new java.util.Random();
  scale = rand.nextInt(N-1) + 1;                    // new hash scaling factor
  shift = rand.nextInt(N);                          // new hash shifting factor
  for (int i=0; i&ltB.length; i++)
    if ((B[i] != null) && (B[i] != AVAILABLE)) { // if we have a valid entry
       int j = findEntry(B[i].key());  // find the appropriate spot
       A[-j-1] = B[i];                 // copy into the new array
    }
}
/** Removes the key-value pair with a specified key. */
public Object remove (Object key) throws InvalidKeyException {
  int i = findEntry(key);               // find this key first
  if (i < 0) return null;               // nothing to remove
  Object toReturn = A[i].value();
  A[i] = AVAILABLE;                                 // mark this slot as
      deactivated
  n--;
  return toReturn;
}
/** Returns an iterator of keys. */
public java.util.Iterator keys() {
  List keys = new NodeList();
  for (int i=0; i&ltN; i++)
    if ((A[i] != null) && (A[i] != AVAILABLE))
       keys.insertLast(A[i].key());
  return keys.elements();
}
} // ... values() is similar to keys() and is omitted here ...
```