INTRODUZIONE

- Una tabella hash è una struttura dati che permette operazioni di ricerca e inserimento molto veloci: in pratica si ha un costo computazionale costante *O(1)*.
- Si ricorda che la ricerca in un *array ordinato* ha un costo *O(log n)*, ma un inserimento/estrazione ha un costo lineare *O(n)*.
- Non è richiesto che gli oggetti memorizzati implementino l'interfaccia `Comparable`, questo perché una tabella hash non è basata sull'operazione di confronto.

INTRODUZIONE

- La realizzazione di una tabella hash è basata sull'uso di array.
- Uno svantaggio è il costo da pagare per ridimensionare la tabella.
- Inoltre non c'è un modo conveniente per visitare gli elementi secondo un certo ordine.
- Tuttavia l'efficienza è tale che esempi di applicazioni includono la *tabella dei simboli* di un compilatore e la ricerca in un *dizionario* di parole chiave.

TABELLE HASH

- Si vogliono memorizzare degli oggetti in una tabella di dimensione `n` (con posizioni da `0` a `n-1`).
- Si memorizzano gli oggetti assegnando ad ogni oggetto un numero intero `i`, detto *codice hash*, compreso tra `0` e `n-1` e inserendo tale oggetto nell'i-esima posizione della tabella.
- Per la ricerca di un oggetto si calcola il suo codice hash e si accede alla corrispondente posizione nella tabella.

FUNZIONE HASH

- La funzione per calcolare il valore hash di un oggetto, detta *funzione hash*, deve soddisfare le seguenti proprietà:
  - Essere semplice, per non influenzare il costo computazionale delle operazioni.
  - Trasformare (*mappare*, *to map*) oggetti uguali in numeri uguali.
- Il senso di uguale è quello del metodo `equals()`.

ESEMPIO

- Si supponga di avere un certo numero di impiegati, per esempio 100, ognuno ha un proprio numero identificativo `idnum` compreso nell'intervallo [0,99].
- Si può pensare di accedere alle informazioni di un impiegato per mezzo della chiave `idnum`. Se si memorizzano i dati in un array di 100 elementi, allora si può accedere alle informazioni di un impiegato direttamente attraverso l'indice dell'array.
- Quindi si ha un costo *O(1)*.

ESEMPIO

- Tuttavia in pratica non è facile avere o mantenere una relazione perfetta tra il valore delle chiavi e gli indici di un array.
- Se l'intervallo del numero identificativo fosse più ampio, per esempio [0,99999], non sarebbe conveniente usare un array di 100000 elementi di cui solo 100 sono necessari.
- È necessario utilizzare un array della corretta dimensione e utilizzare solo due cifre della chiave per identificare un impiegato: per esempio l'impiegato `21374` è in `array[74]` e l'impiegato `32821` è in `array[21]`.

ESEMPIO

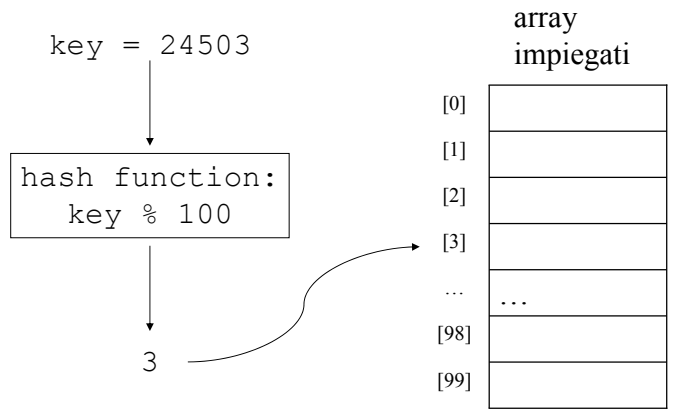
- In tal modo gli elementi *non* risultano ordinati secondo il valore della chiave (l'impiegato 21374 dovrebbe precedere l'impiegato 32821), ma secondo una *certa funzione del valore della chiave*.
- Tale funzione è detta funzione hash (*hash function*) e la tecnica di ricerca è detta *hashing*. La struttura sottostante è detta tabella hash (*hash table*).
- Nell'esempio la funzione hash è:

```
hashValue = key % 100;
```

- La chiave è divisa per la dimensione della tabella e il resto della divisione (`%`) è utilizzato come indice nell'array.

ESEMPIO

- L'estrazione e l'inserimento dei dati sono implementati come semplici operazioni su array attraverso indici. Rappresentazione grafica.



FUNZIONE HASH

- Se la chiave non è numerica (per esempio una stringa, parole di un dizionario) si deve trasformare tale chiave in una forma numerica.
- Vediamo il caso generale delle stringhe.
- Si può pensare di combinare i caratteri (il loro valore numerico) della stringa per avere una rappresentazione numerica.
- Supponiamo di avere un dizionario con 50000 vocaboli, ognuno composto da non più di 10 caratteri.

FUNZIONE HASH

- Un approccio semplice è quello di sommare il valore dei caratteri: per esempio la stringa `casa` produce il codice `3+1+19+1=24`. I valori dei caratteri sono `a=1`, `b=2`, `c=3` …
- Se si considera la stringa più corta (una `a`) e quella più lunga (dieci `z`) si ottiene che l'intervallo dei codici è [1,260], si suppongono 26 caratteri diversi.
- Con questo approccio si ottengono troppe collisioni, cioè *troppe parole associate allo stesso codice*. Si deve cercare una funzione che distribuisca meglio le parole.

FUNZIONE HASH

- Un modo differente di "mappare" parole in numeri è quello di fare in modo che ogni carattere contribuisca in modo unico al numero finale.
- Si segue un approccio polinomiale, utilizzando lo stesso principio di rappresentazione in base dieci dei numeri.
- Le lettere sono moltiplicate per un'appropriata potenza di 26: l'esempio precedente per la stringa `casa` produce il seguente codice $3*26^3+1*26^2+19*26^1+1*26^0=53899$.

FUNZIONE HASH

- Questa tecnica produce un numero unico per ogni potenziale parola. Tuttavia l'intervallo dei codici è troppo ampio, circa 26^{10} che vale $\sim 1.4*10^{14}$; non è memorizzabile.
- Inoltre in tal modo si assegnano codici a parole che non esistono (per esempio, abababababab).
- È necessario *comprimere* questo intervallo: si usa l'operatore *modulo* (`%`) la dimensione della tabella.
- Quindi la funzione hash torna un valore a cui viene applicato l'operatore modulo per ottenere un indice della tabella.
- Tuttavia in tal modo si producono delle collisioni.

FUNZIONE HASH

- Per ogni oggetto il metodo che implementa la funzione hash è ereditato dalla classe `Object` e pertanto deve essere ridefinito (*overridden*).
- In particolare se due oggetti confrontati col metodo `equals()` risultano uguali, allora il metodo `hashCode()` deve ritornare lo stesso valore.
- Scrivere una funzione hash corretta è "facile" (deve tornare lo stesso intero per oggetti uguali), è "difficile" scrivere una funzione efficiente (deve fornire una buona distribuzione degli oggetti).

VALUTAZIONE DI POLINOMI

- Una implementazione immediata per la valutazione dei polinomi implica un calcolo diretto di tutte le potenze x^n, questo produce un costo computazionale *quadratico*.

- Si può utilizzare l'algoritmo di Horner che ha costo *lineare*. Tale algoritmo si basa sull'uso delle parentesi:

$a_4x^4+a_3x^3+a_2x^2+a_1x+a_0=(((a_4x+a_3)x+a_2)x+a_1)x+a_0$

VALUTAZIONE DI POLINOMI

- Per evitare di avere numeri troppo grandi sarebbe necessario fornire alla funzione di hash la dimensione della tabella.
- Altrimenti si lascia che la funzione hash torni numeri negativi e si gestiscono al momento dell'operazione modulo.

- Vediamo un esempio per le stringhe (in realtà la classe `String` implementa già il metodo, quindi si può usare quello).

VALUTAZIONE DI POLINOMI

```
public int hashCode(){
    int hashVal = 0;
    for(int i=0; i<str.length();i++)
          hashVal = 26*hashVal + str.charAt(i);
    return hashVal;
}
```

Nome di un campo stringa

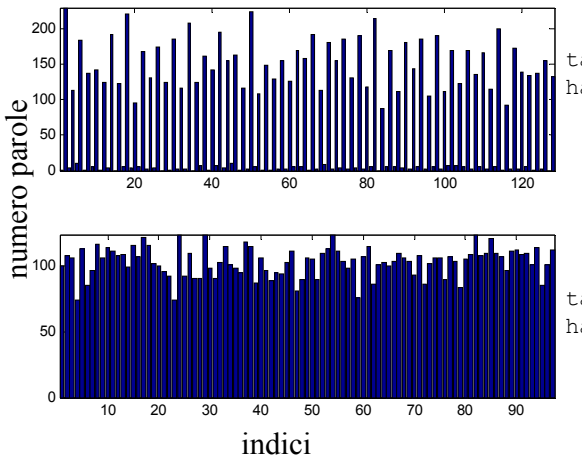
Per usare il valore ritornato si può scrivere:

```
int tablesize = 100;
int index = "provaduetre".hashCode();
index = index % tablesize;
if (index<0)
    index+=tablesize;
```

NUMERI PRIMI

- Per avere una buona distribuzione degli oggetti nella tabella (avere poche collisioni) è conveniente utilizzare dei *numeri primi* per la base delle potenze e per la dimensione delle tabelle.
- Se molte chiavi condividono un divisore con la dimensione della tabella, allora tendono ad utilizzare la stessa posizione.
- Vediamo un esempio: i grafici rappresentano la distribuzione di 10000 parole distinte con due diverse scelte per la dimensione della tabella e per la base del polinomio.

NUMERI PRIMI




```
tablesize = 128;
hashVal = 26*hashVal + …;
```

```
tablesize = 97;
hashVal = 31*hashVal + …;
```


NUMERI PRIMI

- Dato un numero `n` si può dimostrare che esiste un numero primo compreso tra `n` e `2n`.
- Per trovare tutti i numeri primi minori di un dato numero si può utilizzare il crivello di Eratostene.
- L'idea di base è la seguente: dato un numero primo, tutti i suoi multipli non sono primi. Se si considera un insieme che contiene tutti i numeri maggiori o uguali a due, ogni volta che si individua un numero primo si deve escludere dall'insieme tutti i suoi multipli.

NUMERI PRIMI

```
static boolean[] primes(int n){
   boolean[] p = new boolean[n];
   for(int i=2; i<n;i++)
        p[i]=true;
   for(int i=2; i<n;i++){
        if (p[i]!=false){
             for(int j=i;(long)j*i<n;j++)
                  p[i*j]=false;
        }
   }
   return p;
}
```

NUMERI PRIMI

- Questo frammento di codice stampa tutti i numeri primi minori di 100:

```
boolean[] p = primes(100);
  for (int i=0; i<p.length; i++)
     if (p[i])
          System.out.print(i+" ");
```

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97

COLLISIONI

- Quando si "mappano" grandi insiemi di oggetti in interi di piccole dimensioni, le collisioni sono inevitabili. È importante che vi sia una distribuzione uniforme.
- Esistono diverse strategie di risoluzione delle collisioni:
  - Indirizzamento aperto (*open-addressing*).
    - Scansione lineare (*linear probing*).
  - Concatenazioni separate (*separate chaining*).

COLLISIONI: indirizzamento aperto

- Se si memorizzano `n` elementi in una tabella di dimensione `m>n` allora si può contare sul fatto di avere *spazio libero* per gestire le collisioni.
- Un semplice metodo ad *indirizzamento aperto* è la *scansione lineare*: quando si verifica una collisione è sufficiente sondare la successiva posizione della tabella per trovare una locazione libera ed eventualmente proseguire sino a tornare alla posizione iniziale se si è giunti alla fine.
- Se il numero di elementi da inserire non è noto allora è preferibile usare la concatenazione separata.

COLLISIONI: concatenazioni separate

- In questo approccio si costruisce una *lista concatenata* per ogni posizione della tabella. Per esempio inserire gli interi `56 31 17 48 96 98 25 92 98 22 58 49` in una tabella di dimensione `5`: