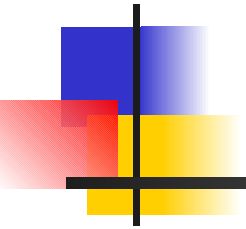
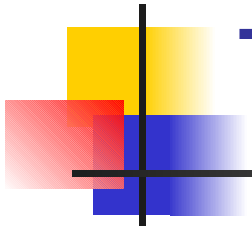


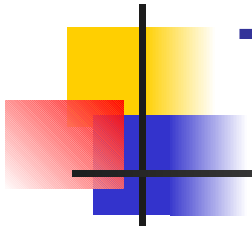
Esercitazione sull' hashing

17 giugno 2004



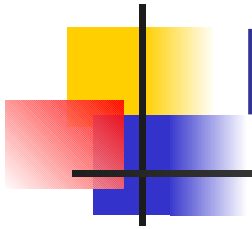
Tipi Wrapper semplici / 1

- I tipi semplici, come int e char, non fanno parte della gerarchia degli oggetti; vengono passati per valore e non possono essere passati per riferimento.
- Alcune classi gestiscono solo oggetti (Vector): abbiamo bisogno di una rappresentazione oggetto dei tipi semplici



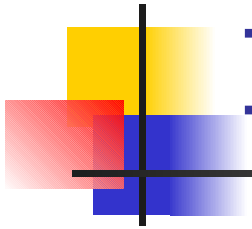
Tipi Wrapper semplici /2

- Java offre classi che corrispondono a ciascuno dei tipi semplici.
- In pratica, queste classi incapsulano (wrap), i tipi semplici all'interno di una classe.
- Tali classi sono definite all'interno di java.lang.



Number

- La classe astratta Number è la superclasse delle classi BigDecimal, BigInteger, Byte, Double, Float, Integer, Long, and Short.
- Le sottoclasse di Number devono fornire metodi per convertire il valore del tipo numerico rappresentato in byte, double, float, int, long, e short.

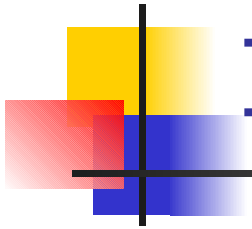


Integer /1

- La classe Integer incapsula il valore del tipo primitivo int in un oggetto. Un oggetto di tipo Integer contiene un unico campo il cui tipo è int.
- Costruttori
  - **Integer**(int value)
  - **Integer**(String s)

Integer /2

- Acuni metodi:
  - byte **byteValue**()
  - int **intValue**();
  - long **longValue**();
  - String **toString**();
- Metodi simili per le classi Float, Double e Long.
- Esistono wrapper anche per le classi char e boolean



Il nostro wrapper: BaseObject / 1

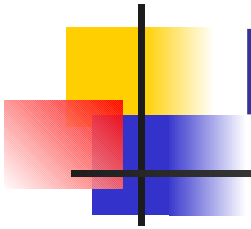
```
 // Wrapper class for use with generic data structures.

public class BaseObject implements Comparable {

  private int key;

public BaseObject(int k) {
    this.setKey(k);
   }
public void setKey(int k) {
    key = k;
    }

   public int intValue(){
     return key;
}
```

Il nostro wrapper: BaseObject / 2

```
public int compareTo(Object bo) {
    if(!(bo instanceof BaseObject))
      throw new ClassCastException();
    if(this.key == ((BaseObject)bo).key)
      return 0;
      else if(this.key < ((BaseObject)bo).key)
        return -1;
            else return 1;
  }

public String toString() {
     return "" + key + " ";
   }
```

Il nostro wrapper: BaseObject / 3

```
public void visit() {
    System.out.print(this.toString());
  }

  /**
   * Implements the equals method.
         * @param anotherBaseObject the second BaseObject.
         * @return true if the objects are equal, false otherwise.
         * @exception ClassCastException if anotherBaseObject is not
         *     a BaseObject.
         */
public boolean equals(BaseObject anotherBaseObject) {
            return this.key == anotherBaseObject.key;
    }
}
```

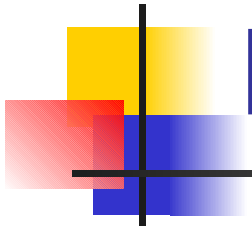


L’interfaccia comparable

- Questa interfaccia impone un ordinamento totale sugli oggetti di ciascuna classe che la implementa.
- Questo ordinamento viene considerato come l’ordinamento naturale della classe ed il metodo compareTo definisce il metodo di confronto.
- I wrapper di tipi numerici in Java implementano l’interfaccia Comparable.
- Anche BaseObject!

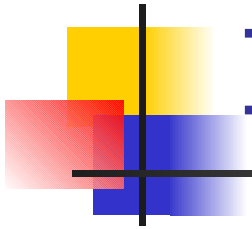
La classe java.util.Random

- La classe Random è un generatore di numeri pseudo-casuali, così definiti perché non sono altro che sequenze uniformemente distribuite.
- Definisce i seguenti costruttori:
  - Random(): crea un generatore di numeri casuali. Il suo seme è inizializzato con un valore basato sul tempo corrente: System.currentTimeMillis()
  - Random (long seme): crea un generatore di numeri casuali usando il seme dichiarato.
- Metodi:
  - int **nextInt**(int n):Restituisce un valore intero uniformemente distribuito tra 0 (incluso) e lo specificato valore n (escluso), a partire dalla sequenza random corrente.



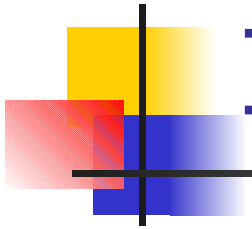
Esercitazione

- Copiare la cartella Esercitazioe3 nella propria directory di lavoro
- Studiare le classi da completare
  - HashTable.java
  - HashTableDemo.java
  - Se serve, alcune delle slide che seguono riassumono l'implementazione di HashTable.java
- Rispondere alle domande
  - V. slide successive



Implementazione Java

- Tabella rappresentata con array
- Chiavi di tipo Comparable
  - Convertite in String
- Funzione hash applicata a oggetti String
  - i = 0: h(S, 0) = h(S)
    - S una stringa
  - i > 0: h(S, i) = h(S) + cost * i
    - cost = 3

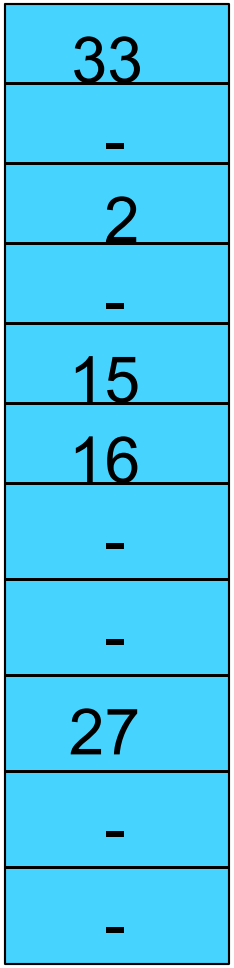


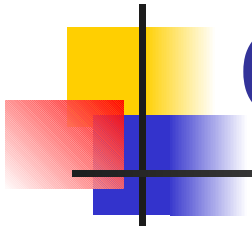
Implementazione Java/cont.

- Due array per rappresentare la tabella:
  - Comparable table[] - entry della tabella
  - Boolean isActive[] - per la gestione
    - isActive[i] = true se table[i] != null oppure table[i] == null, ma table[i] ha precedentemente contenuto un elemento
- Problema: efficienza
  - I metodi di ricerca e cancellazione visti precedentemente possono essere poco efficienti
    - Es.: se elemento assente viene scandita l'intera tabella

Esempio: ricerca

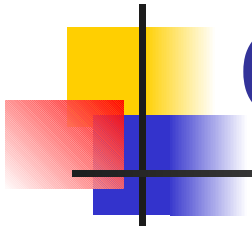
- Scansione lineare
  - h(k, i) = k mod 11 + 3*i
- Ricerca chiave 27
  - Posizione (5) occupata da elemento di chiave 16
  - Come facciamo a sapere che possiamo interrompere la ricerca?
- Necessario riconoscere le posizioni che sono state occupate da elementi poi rimossi





Classe HashTable

```
public class HashTable implements Dictionary_adt {
    private static final int DEFAULT_TABLE_SIZE = 23;
     /** The array of elements. */
    protected Comparable [ ] table;    // The array of elements
    protected boolean [] isActive;
    protected int currentSize;       // The number of occupied cells
    protected boolean isRehashable;  //true if table can be expanded
    protected int k =3;              //Coefficient for LinearProbing

    /* Seguono costruttori */
```

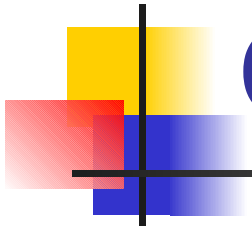


Classe HashTable/2

```
/* Costruttore principale */
public HashTable(int size, boolean rehash) {
/* Alloca due array table[] e isActive[] di simensione
size. Se rehash == true -> la tabella puo' essere
espansa se e' piena per piu' della meta'.
Inizializza la tabella: table[i] = null e
isActive[i] = false per ogni i */
}
/* Altri costruttori */

/* Seguono i metodi */
```

Classe HashTable/3

```
/* La nostra funzione hash */
public static int hash(String key, int tableSize) {
int hashVal = 0;

        for( int i = 0; i < key.length( ); i++ )
                hashVal = 37 * hashVal + key.charAt( i );
        hashVal %= tableSize; /* | hashVal | < tableSize */
        if( hashVal < 0 ) /* si puo' avere overflow -> hashVal < 0 */
                hashVal += tableSize; /* Cosi' hashVal diventa > 0 */
        return hashVal;
}
/* Altri metodi */
```



Classe HashTable/4

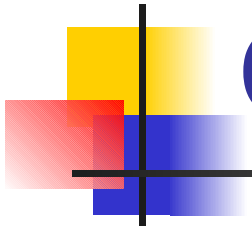
```
/* Inserimento */
public Object insert(Comparable key ) {
int collisionNum = 0; /* No. collisioni */
int initialPos = hash( key.toString(),table.length );
int currentPos=initialPos;
int insertPos= -1; /* Se alla fine vale -1 -> tabella piena
                      (caso di tabella non espandibile) */

/* Continua ... */
```

Classe HashTable/5

```
while(collisionNum<table.length-1) {
        if (table[currentPos] == null) { /* trovata posizione libera */
                if (insertPos == -1)
                        insertPos = currentPos;
                if (!isActive[currentPos])
                        break; /* se posizione non attiva -> elemento non presente,
                                puoi uscire dal ciclo while */
        } /* End if */
        else if (table[ currentPos ].compareTo( key )==0) {
        //table[currentPos]!= null
                System.out.println("Element "+ key +" is alredy in the hash table.");
                return key;
        } /* End else */
        currentPos = initialPos + k * ++collisionNum;
        currentPos = currentPos % table.length; // Implement the mod
} /* End while */
```

Classe HashTable/6

```
 if (insertPos!= -1) { /* E' stata trovata una posizione libera */
        table[ insertPos ] = key;
        isActive[ insertPos] = true;
        ++currentSize;
        System.out.println("Insertion of "+key+": in position " +currentPos);
        if((isRehashable)&&(currentSize > table.length/2)) {
                System.out.println("Rehash!");
                rehash();
        } /* End if  */
   return key;
   } /* End if */
   else { /* Non e' stata trovata una posizione libera */
        System.out.println("Insertion impossible: hash table full");
        return null;
   } /* End else */
} /* End inserimento */
```



Classe HashTable/7

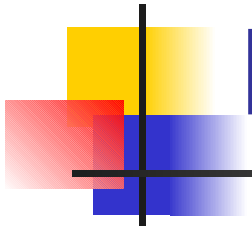
```
public Comparable remove(Comparable key ){
int collisionNum = 0;
int initialPos = hash( key.toString(),table.length );
int currentPos=initialPos;

/* Variabili hanno stesso significato che in insert() */
/* Continua */
```

Classe HashTable/8

```
while ( ( ((table[currentPos] == null)&& isActive[currentPos] )||
( (table[currentPos] != null) &&
(table[currentPos].compareTo(key)!=0))) &&
(collisionNum < table.length-1)) {
        currentPos = initialPos + k * ++collisionNum; // Compute ith
probe
        currentPos = currentPos % table.length; // Implement the mod
} /* End while */
/* Esce da ciclo while se elemento trovato o tutta la tabella esaminata
senza trovare elemento con chiave cercata */
/* Continua */
```

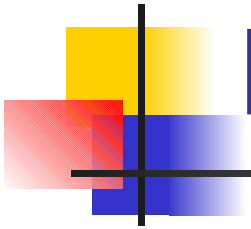
Classe HashTable/9

```
    /* Continua dalla slide precedente */
    if ((table[currentPos]==null)||table[currentPos].compareTo(key) !=0) {
            System.out.println("Element "+ key +" isn't in the hash table.");
            return null;
    } /* End if */
    else{
            System.out.println("Element "+ key +" is in the hash table.");
            table[currentPos]=null;
            return key;
    } /* End else */
} /* End remove() */
/* Altri metodi e fine della classe HashTable */
```



Domande-parte prima

- Completare i metodi seguenti di HashTable.java:
  - public void startCounter(): inizializza (pone a 0) un contatore del numero di collisioni
  - public int getColl(): restituisce il numero di collisioni registrate a seguito di inserimenti e a partire dall'ultima volta che il contatore e' stato inizializzato
    - Si osservi che il numero di collisioni puo' essere maggiore del numero di inserimenti, perche' ogni elemento puo' collidere piu' volte
  - public int maxClusterSize(): restituisce la dimensione del cluster piu' grande
    - Cluster definiti nelle slide seguenti

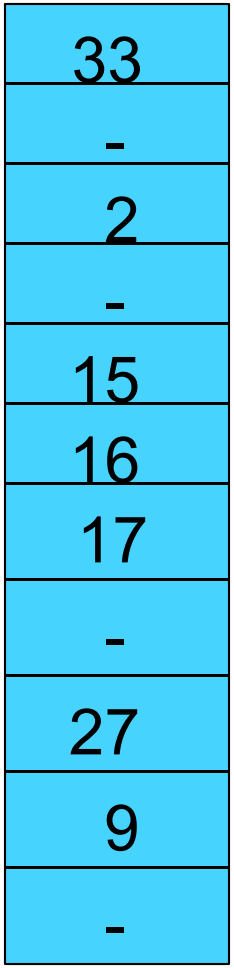


Domande-parte prima/2

- Modificare HashTableDemo.java in modo che sia stampata anche la percentuale delle collisioni sul totale degli inserimenti
- Raccomandazione: provare il funzionamento dei metodi di volta in volta implementati, usando la classe HashTableDemo

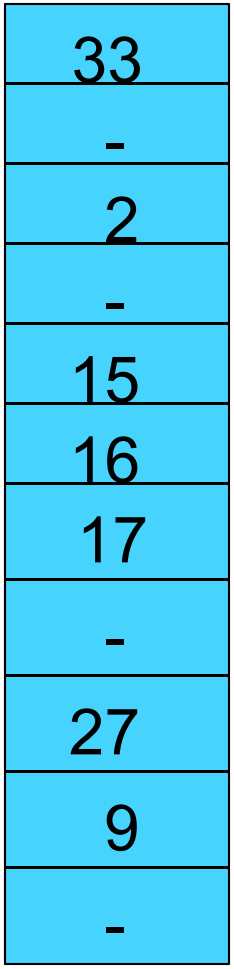
Clustering

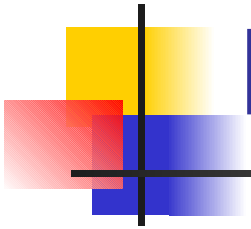
- Cluster: sequenza di posizioni consecutive non vuote
- I cluster si formano per effetto dell'agglomerazione primaria e secondaria



Clustering/2

- Esempio: le posizioni 4, 5 e 6 della tabella formano un cluster di dimensione 3, mentre le posizioni 8 e 9 formano un cluster di dimensione 2
- Massima dimensione cluster = 3
- Fattore di clustering = (max. dim. cluster)/(Dim. tabella)
  - 3/11 in questo esempio
- Se la tabella e' espandibile il fattore di clustering misura l'efficacia della funzione di hash nel distribuire le chiavi in modo uniforme





Domande/parte seconda

- Modificare il metodo insert() di Hashtable.java in modo che la dimensione della tabella sia raddoppiata quando una frazione frac (0 < frac < 1) della tabella e' occupata
  - Nella implementazione corrente frac = 0.5
  - frac puo' essere un campo static final di HashTable.java
- Eseguire HashTableDemo con 3 valori diversi di frac
  - Es.: frac = 0.5, 0.7 e 0.9
  - Come varia il fattore di clustering al variare di frac?