Metodologie di Programmazione

Lezione 23: Le collezioni (parte 2)

Lezione 23: Sommario

UNITELMA SAPIENZA
SAPIENZA UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

- Mappe
  - HashMap
  - TreeMap
  - LinkedHashMap
- Chiavi e valori
- Algoritmi su collezioni e array
- Ordinamento
  - Comparable e Comparator
- Pile e code
- Alberi

Mappe

- Una mappa mette in corrispondenza chiavi e valori
- **Non** può contenere chiavi duplicate
- java.util.Map è un'interfaccia implementata da HashMap e TreeMap

L'interfaccia java.util.Map

Method Summary

Methods

Modifier and Type	Method and Description
void	`clear()` Removes all of the mappings from this map (optional operation).
boolean	`containsKey(Object key)` Returns true if this map contains a mapping for the specified key.
boolean	`containsValue(Object value)` Returns true if this map maps one or more keys to the specified value.
`Set<Map.Entry<K,V>>`	`entrySet()` Returns a `Set` view of the mappings contained in this map.
boolean	`equals(Object o)` Compares the specified object with this map for equality.
`V`	`get(Object key)` Returns the value to which the specified key is mapped, or `null` if this map contains no mapping for the key.
int	`hashCode()` Returns the hash code value for this map.
boolean	`isEmpty()` Returns true if this map contains no key-value mappings.
`Set<K>`	`keySet()` Returns a `Set` view of the keys contained in this map.
`V`	`put(K key, V value)` Associates the specified value with the specified key in this map (optional operation).
void	`putAll(Map<? extends K,? extends V> m)` Copies all of the mappings from the specified map to this map (optional operation).
`V`	`remove(Object key)` Removes the mapping for a key from this map if it is present (optional operation).
int	`size()` Returns the number of key-value mappings in this map.
`Collection<V>`	`values()` Returns a `Collection` view of the values contained in this map.

Mappe

- Una mappa mette in corrispondenza chiavi e valori
- **Non** può contenere chiavi duplicate
- java.util.Map è un'interfaccia implementata da HashMap e TreeMap
- HashMap memorizza le coppie in una tabella di hash
- TreeMap memorizza le coppie in un albero mantenendo un ordine sulle chiavi
- LinkedHashMap estende HashMap e mantiene l'ordinamento di iterazione secondo gli inserimenti effettuati

HashMap: Un esempio

- Registriamo le frequenze di occorrenza delle parole in un testo:

```
public class MappaDelleFrequenze
{
    private Map<String, Integer> frequenze = new HashMap<String, Integer>();

    public MappaDelleFrequenze(File file) throws IOException
    {
        Scanner in = new Scanner(file);
        while(in.hasNext())
        {
            String parola = in.next();
            Integer freq = frequenze.get(parola);

            // parola mai incontrata (non presente nella mappa)
            if (freq == null)   frequenze.put(parola, 1);
            // incrementa il conteggio di frequenza della parola
            else                frequenze.put(parola, freq+1);
        }
    }
}
```

TreeMap: Un esempio

- Se vogliamo mantenere un ordinamento sulle chiavi (ovvero sulle parole) di cui calcoliamo la frequenza:

```
public class MappaDelleFrequenze
{
    private Map<String, Integer> frequenze = new TreeMap<String, Integer>();

    public MappaDelleFrequenze(File file) throws IOException
    {
        Scanner in = new Scanner(file);
        while(in.hasNext())
        {
            String parola = in.next();
            Integer freq = frequenze.get(parola);

            // parola mai incontrata (non presente nella mappa)
            if (freq == null)   frequenze.put(parola, 1);
            // incrementa il conteggio di frequenza della parola
            else                frequenze.put(parola, freq+1);
        }
    }
}
```

TreeMap mantiene l'ordinamento delle chiavi

LinkedHashMap: Un esempio

- Se vogliamo mantenere un ordinamento di iterazione (ovvero sulle parole) di cui calcoliamo la frequenza:

```
public class MappaDelleFrequenze
{
    private Map<String, Integer> frequenze = new LinkedHashMap<String, Integer>();

    public MappaDelleFrequenze(File file) throws IOException
    {
        Scanner in = new Scanner(file);
        while(in.hasNext())
        {
            String parola = in.next();
            Integer freq = frequenze.get(parola);

            // parola mai incontrata (non presente nella mappa)
            if (freq == null)   frequenze.put(parola, 1);
            // incrementa il conteggio di frequenza della parola
            else                frequenze.put(parola, freq+1);
        }
    }
}
```

LinkedHashMap mantiene l'ordinamento di iterazione sulle chiavi

Confronto tra HashMap, TreeMap e LinkedHashMap

- Dato il seguente file:

```
a a e b c d e e b
```

- Se iteriamo su ciascuna delle tre strutture:

```
String s = "{";

for (String chiave : frequenze.keySet())
    s += chiave+"="+frequenze.get(chiave)+", ";

s += "}";
```

Senza ordinamento

Ordinamento naturale

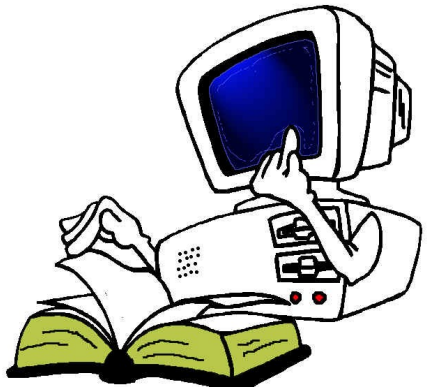
Ordinamento di inserimento

otteniamo:

- HashMap: {d=1, e=3, b=2, c=1, a=2, }
- TreeMap: {a=2, b=2, c=1, d=1, e=3, }
- LinkedHashMap: {a=2, e=3, b=2, c=1, d=1, }

Chiavi e valori di una mappa

- E' possibile ottenere l'insieme delle chiavi di una mappa mediante il metodo keySet
- L'elenco dei valori mediante il metodo values **(con ripetizione!)**
- L'insieme delle coppie (chiave, valore) mediante il metodo entrySet
  - Restituisce un insieme di oggetti di tipo Map.Entry<K, V>
  - Per ogni oggetto (ovvero la coppia) è possibile conoscere la chiave (getKey()) e il valore (getValue())



Un esempio di creazione di una TreeMap

- Il seguente codice crea una TreeMap e ne stampa l'insieme delle chiavi, la collezione dei valori e la collezione delle coppie (di tipo Map.Entry<K, V>):

```
Map<String, Integer> m = new TreeMap<String, Integer>();

m.put("a", 1);
m.put("b", 1);
m.put("c", 2);
m.put("d", 3);
m.put("e", 1);

System.out.println(m.keySet());
System.out.println(m.values());
System.out.println(m.entrySet());
```

- Output:

```
[a, b, c, d, e]
[1, 1, 2, 3, 1]
[a=1, b=1, c=2, d=3, e=1]
```

Algoritmi sulle collezioni: la classe Collections

- Fornisce metodi statici per la manipolazione delle collezioni:

Metodo	Descrizione
sort	Ordina gli elementi di una List
binarySearch	Cerca un elemento di una List mediante ricerca binaria
fill	Rimpiazza tutti gli elementi di una List con l'elemento specificato
copy	Copia gli elementi da una lista all'altra
reverse	Inverte l'ordine degli elementi di una List
shuffle	Mette in ordine casuale gli elementi di una List
min/max	Restituisce l'elemento più piccolo/grande della Collection

Algoritmi sugli array: la classe Arrays

- Fornisce metodi statici per la manipolazione degli array:

Metodo	Descrizione
sort	Ordina gli elementi dell'array
binarySearch	Cerca un elemento mediante ricerca binaria
fill	Riempie l'array con l'elemento specificato
copyOf	Restituisce una copia di un array
equals	Confronta due array elemento per elemento
asList	Restituisce una lista contenente gli elementi dell'array
toString	Restituisce una rappresentazione dell'array sotto forma di stringa

Ordinamento “naturale”

- Come garantire un ordinamento sui tipi utilizzati nelle strutture dati che si fondano su un ordinamento (es. TreeSet o TreeMap)?
- E’ necessario che quei tipi implementino un’interfaccia speciale, chiamata Comparable<T>
- Dotata di un solo metodo:

Metodo	**Descrizione**
int **compareTo**(T o)	Confronta sé stesso con l’oggetto o (restituendo 0 se uguali, -1 se < o, +1 altrimenti)

L'interfaccia Comparable: un esempio

- Come garantire l'ordinamento di una classe NomeCognome?

```
public class NomeCognome implements Comparable<NomeCognome>
{
    private String nome;
    private String cognome;

    public NomeCognome(String nome, String cognome)
    {
        this.nome = nome;
        this.cognome = cognome;
    }

    @Override
    public int compareTo(NomeCognome o)
    {
        int v = cognome.compareTo(o.cognome);

        if (v == 0) return nome.compareTo(o.nome);
        else return v;
    }

    @Override
    public String toString() { return cognome+" "+nome; }
}
```

Ordinamento con l'interfaccia Comparator<T>

- E se la classe non fornisce "di suo" l'ordinamento mediante l'interfaccia Comparable?
- E se voglio ordinare diversamente gli elementi di un certo oggetto?
- E' sufficiente implementare un'interfaccia Comparator<T> e passarne un'istanza in input al costruttore delle strutture dati (es. TreeSet e TreeMap)
- Metodi dell'intefaccia Comparator<T>

Method Summary	
int	compare(T o1, T o2) Compares its two arguments for order.
boolean	equals(Object obj) Indicates whether some other object is "equal to" this comparator.

Pila e coda

- Due strutture dati fondamentali utili in un gran numero di attività
- Coda (FIFO: First In, First Out)

- Pila o stack (LIFO: Last In, First Out)

Coda

- Esempi di coda:
  - Coda degli eventi relativi a mouse e tastiera
  - Coda di stampa
- Esistono implementazioni standard della coda mediante l'interfaccia Queue
  - LinkedList implementa l'interfaccia Queue!
- Operazioni principali:
  - add: inserisce un elemento in coda
  - remove: rimuove un elemento dall'inizio della coda
  - peek: restituisce l'elemento all'inizio della coda senza rimuoverlo

Pila

- Esempi di pila:
  - La pila di esecuzione (run-time stack) contenente i record di attivazione delle chiamate a metodi
  - Nell'implementazione della ricorsione…
- Esiste un'implementazione standard mediante la classe Stack
  - Implementa l'interfaccia List
- Operazioni principali:
  - push: inserisce un elemento in cima alla pila
  - pop: rimuove l'elemento in cima alla pila
  - peek: restituisce l'elemento in cima alla pila senza rimuoverlo

Esercizio: parentesi annidate

- Scrivere un metodo che, data una stringa in input, verifichi se le parentesi della stringa sono correttamente annidate
- Sono ammessi due tipi di parentesi: tonde e quadre

Esercizio: parentesi annidate

- Scrivere un metodo che, data una stringa in input, verifichi se le parentesi della stringa sono correttamente annidate
- Sono ammessi due tipi di parentesi: tonde e quadre

```
import java.util.Stack;

public class Parentesi
{
    public boolean annidate(String s)
    {
        Stack<Character> p = new Stack<Character>();

        for (int k = 0; k < s.length(); k++)
        {
            char c = s.charAt(k);









        }

        return p.isEmpty();
    }
}
```

Esercizio: parentesi annidate

- Scrivere un metodo che, data una stringa in input, verifichi se le parentesi della stringa sono correttamente annidate
- Sono ammessi due tipi di parentesi: tonde e quadre

```
import java.util.Stack;

public class Parentesi
{
    public boolean annidate(String s)
    {
        Stack<Character> p = new Stack<Character>();

        for (int k = 0; k < s.length(); k++)
        {
            char c = s.charAt(k);
            switch(c)
            {
                case '(': case '[': p.push(c); break;
                case ')': if (p.isEmpty() || p.pop() != '(') return false; break;
                case ']': if (p.isEmpty() || p.pop() != '[') return false; break;
            }
        }

        return p.isEmpty();
    }
}
```

Esercizio: inversione degli elementi

- Scrivere un metodo che, data una coda in input, ne inverta gli elementi
- Scrivere la variante in cui viene fornita in input (e restituita in output) una stringa

Esercizio: inversione degli elementi

- Scrivere un metodo che, data una coda in input, ne inverta gli elementi
- Scrivere la variante in cui viene fornita in input (e restituita in output) una stringa

```
public void inverti(Queue<Integer> q)
{
    Stack<Integer> s = new Stack<Integer>();

    while(!q.isEmpty())
    {
        Integer e = q.remove();
        s.push(e);
    }

    while(!s.isEmpty()) q.add(s.pop());
}
```

Esercizio: calcolatrice in notazione polacca inversa

- Scrivere un metodo che, data una lista in input con operandi e operatori in notazione polacca inversa, calcoli il risultato delle operazioni
- Notazione polacca inversa: gli operatori vengono **DOPO** gli operandi (ad esempio: 3 4 + 5 - denota 3+4-5)

Esercizio: calcolatrice in notazione polacca inversa

- Scrivere un metodo che, data una lista in input con operandi e operatori in notazione polacca inversa, calcoli il risultato delle operazioni
- Notazione polacca inversa: gli operatori vengono **DOPO** gli operandi (ad esempio: 3 4 + 5 - denota 3+4-5)

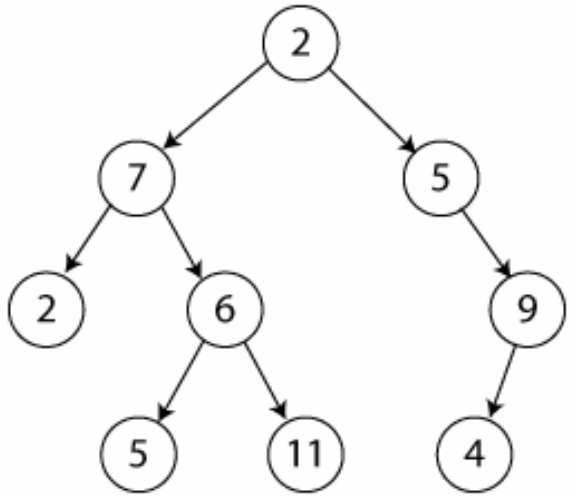
```
public int calcola(List<String> q) throws CalcolatriceException
{
    Stack<Integer> p = new Stack<Integer>();

    for (String v : q)
    {
        if (v.equals("+"))
            p.push(p.pop()+p.pop());
        else if (v.equals("-"))
        {
            Integer arg2 = p.pop();
            p.push(p.pop()-arg2);
        }
        else
        {
            Integer i = Integer.parseInt(v);
            p.push(i);
        }
    }

    // e' rimasto solo il risultato sulla pila?
    if (p.size() == 1) return p.pop();
    else
    {
        // la pila e' ancora piena
        throw new CalcolatriceException();
    }
}
```

Gli alberi

- Una struttura dati ricorsiva in cui ogni nodo possiede un padre tranne la radice
- Gli alberi più comuni sono binari (ovvero con al più due figli per nodo)



Rappresentare gli alberi in Java

- Utilizziamo una classe interna per rappresentare il nodo di un albero binario:

```java
/**
 * Rappresenta un albero binario
 */
public class BinaryTree
{
    /**
     * Radice dell'albero
     */
    private Node root;

    public class Node
    {
        private Node left;
        private Node right;
        private int valore;

        private Node(Node left, Node right, int valore)
        {
            this.left = left;
            this.right = right;
            this.valore = valore;
        }

        public void setLeft(Node left) { this.left = left; }
        public void setRight(Node right) { this.right = right; }
        public Node getLeft() { return left; }
        public Node getRight() { return right; }
        public int getValore() { return valore; }
    }
}
```

Ricerca all'interno di un albero binario

- Come ricercare un'informazione all'interno di un albero binario?

```java
/**
 * Rappresenta un albero binario
 */
public class BinaryTree
{
    /**
     * Radice dell'albero
     */
    private Node root;

    public class Node
    {
        private Node left;
        private Node right;
        private int valore;

        private Node(Node left, Node right, int valore)
        {
            this.left = left;
            this.right = right;
            this.valore = valore;
        }

        public void setLeft(Node left) { this.left = left; }
        public void setRight(Node right) { this.right = right; }
        public Node getLeft() { return left; }
        public Node getRight() { return right; }
        public int getValore() { return valore; }
    }

    public boolean contains(int val)
    {
        return contains(root, val);
    }

    public boolean contains(Node n, int val)
    {
        if (n == null) return false;
        if (n.valore == val) return true;

        return contains(n.left, val) || contains(n.right, val);
    }
}
```