Alberi binari

Definizione di albero binario:

matematica:
: un albero è:
  - un albero vuoto, oppure
  - è composto da un dato più due alberi

in Java:
: un oggetto che contiene due riferimenti a due oggetti dello stesso tipo

---

Rappresentazione

semplificata:
: gli alberi sono oggetti con un campo `info` e due campi `sinistro` e `destro`

object-oriented:
: rappresentazione incapsulata di un albero (in senso matematico)

Rappresentazione semplificata

Tutti campi pubblici

```
class Albero {
  public Object info;
  public Albero sinistro;
  public Albero destro;

  public Albero(Object i, Albero s, Albero d) {
    info=i;
    this.sinistro=s;
    this.destro=d;
  }
}
```

Il costruttore non è necessario

Ricorsione

Si parte dal presupposto che il metodo funzioni

Si scrive il corpo del metodo

Lasciate perdere i record di attivazione (per ora)

Esempio

Stampa degli elementi di un albero

Passo 1

si scrive l'intestazione:

```
static void stampaTutti(Albero a) {
  // corpo della funzione
}
```

Assunzione ricorsiva

```
static void stampaTutti(Albero a) {
  // corpo della funzione
}
```

Nel corpo del metodo si può usare la funzione `stampaTutti`

```
static void stampaTutti(Albero a) {
  // corpo della funzione
  ...
  stampaTutti(b);
  ...
}
```

L'invocazione `stampaTutti(b)` dentro il corpo funziona a condizione che `b` sia più piccolo di `a`

Nel caso specifico: si può assumere che `stampaTutti(b)` produca la stampa di tutti i nodi di `b`

Passo 2

Nel corpo, si possono scrivere sia `stampaTutti(a.sinistro)` che `stampaTutti(a.destro)`

Si può fare perchè sia `a.sinistro` che `a.destro` sono più piccoli di `a`

```
static void stampaTutti(Albero a) {
  ...
  stampaTutti(a.sinistro);
  ...
  stampaTutti(a.destro);
  ...
}
```

Si puè assumere che siano corrette, ossia che vengano stampati gli elementi dei sottoalberi

---

Passo 3

Si completa il metodo

```
static void stampaTutti(Albero a) {
  if(a==null)
    return;

  stampaTutti(a.sinistro);
  stampaTutti(a.destro);
  System.out.print(a.info+" ");
}
```

Bisogna tenere conto che `a` può valere `null`

Dopo aver stampato gli elementi del sottoalbero sinistro e quelli del destro, va stampata la radice

---

Assunzione ricorsiva e record di attivazione

Si usano in contesti diversi:

assunzione ricorsiva
: si usa per progettare il metodo (assumo che le invocazioni ricorsive siano corrette)

record di attivazione
: sono il meccanismo che si usa nel linguaggio di programmazione per realizzare la ricorsione

Progettare i metodi pensando ai record di attivazione è uno sbaglio

La progettazione si fa assumendo che, dentro il corpo del metodo, una invocazione ricorsiva fa quello che dovrebbe se il problema è stato semplificato (es. si passa un albero più piccolo)

Visite

I quattro tipi di visite "standard" di un albero binario:

preordine
: si visita la radice e poi i sottoalberi

postordine
: prima i sottoalberi e poi la radice

simmetrica
: prima il sottoalbero sinistro, poi la radice e poi il sottoalbero destro

per livelli (detta anche visita in ampiezza)
: il livello di un nodo è la sua distanza dalla radice; si stampa prima il livello 0, poi il livello 1, ecc.

Visite in profondità

Le visite in preordine, postordine e simmetrica si dicono anche "in profondità"

Implementazione:

```
static void preordine(Albero a) {
  if(a==null)
    return;

  System.out.print(a.info+" ");
  preordine(a.sinistro);
```

```
  preordine(a.destro);
}

static void postordine(Albero a) {
  if(a==null)
    return;

  postordine(a.sinistro);
  postordine(a.destro);
  System.out.print(a.info+" ");
}

static void simmetrica(Albero a) {
  if(a==null)
    return;

  simmetrica(a.sinistro);
  System.out.print(a.info+" ");
  simmetrica(a.destro);
}
```

Visita=stampa?

I metodi di visita dicono in che ordine si deve effettuare una certa operazione

L'operazione può essere qualsiasi

Esempio: stampa il campo `info` di un nodo solo sè è un intero di valore `<4`

```
static void stampaMinori(Albero a) {
  if(a==null)
    return;

  Integer i=(Integer) a.info;

  if(i.intValue()<4)
    System.out.print(i.intValue()+" ");

  stampaMinori(a.sinistro, limite);
  stampaMinori(a.destro, limite);
}
```

Metodi parametrici

Se si vogliono stampare solo gli elementi *maggiori* di 4?

Si vogliono stampare i campi `info` che sono di tipo `String`, ma al massimo i primi 10 caratteri?

Servirebbe poter rendere parametrico un metodo:

```
static void operaTutti(Albero a, "operazione") {
  if(a==null)
    return;

  "esegui operazione su a.info";

  operaTutti(a.sinistro);
  operaTutti(a.destro);
}
```

Sto cercando di dire al metodo `operaTutti` che deve eseguire una operazione specificata come parametro, non la stampa

---

Passare un metodo

A un metodo non si può passare un altro metodo, come si può fare in C.

Posso scrivere un metodo che fa la stampa condizionale:

```
// corretto
static void stampaSeMinore4(Object o) {
  Integer i=(Integer) o;

  if(i.intValue()<4)
    System.out.println(i.intValue());
}
```

Ora vorrei dire al metodo `preordine` di usare il metodo `stampaSeMinore` invece di `println`

Ma in Java non si può passare ad un metodo un altro metodo, come si può fare in C:

```
// errore
static void preordine(Albero a, metodo) {
  if(a==null)
    return;

  metodo(a.info);
  preordine(a.sinistro, metodo);
  preordine(a.destro, metodo);
}
```

Oggetti funzione

A un metodo si può passare soltanto uno scalare oppure un oggetto

Soluzione: passo un oggetto soltanto per per poter passare il metodo

---

Meccanismo degli oggetti funzione

1. si scrive una classe con la funzione che va invocata ad ogni passo:

```
class Funzione4 {
  public void azione(Object o) {
    Integer i=(Integer) o;
    if(i.intValue()<=4)
      System.out.print(i.intValue()+" ");
  }
}
```

2. si modifica il metodo in modo tale che:
   - abbia come parametro un oggetto
   - invochi il metodo sull'oggetto invece di fare `println`

```
static void preordine(Albero a, Object f) {
  if(a==null)
    return;

  f.azione(a.info);
  preordine(a.sinistro, f);
  preordine(a.destro, f);
}
```

3. quando si invoca il metodo `preordine`, si passa un oggetto della classe:

```
preordine(a, new Funzione4());
```

Non può funzionare, ma questo lo vediamo dopo.

Come funziona

Il metodo viene chiamato usando `preordine(a, new Funzione4());`

Nel metodo `preordine`, viene invocato `f.azione`

Questo è il metodo `azione` della classe `Funzione4`

Altre operazioni sui nodi

Se voglio fare una operazione diversa su tutti i nodi dell'albero: scrivo un'altra classe al posto di `Funzione4`

```
class FunzioneString {
  static void azione(Object o) {
    if(o instanceof String)
      System.out.println(o);
  }
}
```

Per far eseguire questo metodo su tutti i nodi dell'albero, basta fare:

```
preordine(a, new FunzioneString());
```

Quando `preordine` invoca `f.azione`, viene eseguito il metodo `azione` della classe `FunzioneString`

Cosa non funziona

```
static void preordine(Albero a, Object f) {
  ...
  f.azione(a.info);
  ...
}
```

La classe `Object` non ha il metodo `azione`

Falsa soluzione

Modifico l'intestazione del metodo `preordine`:

```
  static void preordine(Albero a, Funzione4 f) {
    if(a==null)
      return;

    f.azione(a.info);
    preordine(a.sinistro, metodo);
    preordine(a.destro, metodo);
  }
```

Ora però non posso più passare al metodo un oggetto della classe `FunzioneString`!

In questo caso, posso solo stampare interi minori di quattro (il metodo non è più parametrico)

Principio generale

Codifico la funzione da eseguire su tutti i nodi dell'albero in una classe:

```
class NomeClasse {
  void azione(Object o) {
    ...
  }
}
```

Modifico il metodo di visita in preordine in modo tale che:

1. riceve come argomento un oggetto `f`
2. esegue `f.azione(a.info)` su tutti i nodi

La classe di `f`

Il metodo ha intestazione:

```
static void preordine(Albero a, Qualcosa f) {
  ...
  f.azione(a.info);
  ...
}
```

La classe `Qualcosa` non può essere:

`Object`
: troppo generico: non ha il metodo `azione`

`Funzione4`
: troppo specifico: si possono solo stampare interi minori di 4!

L'oggetto `f` deve essere istanza di una classe che ha il metodo `azione(Object o)`

Come si impone la presenza di un metodo in una classe?

L'interfaccia `FunzioneVoid`

Specifica dell'intefaccia:

```
interface FunzioneVoid {
  public void azione(Object o);
}
```

Ora posso scrivere il metodo `preordine`:

```
  static void preordine(Albero a, FunzioneVoid f) {
    if(a==null)
      return;

    f.azione(a.info);
    preordine(a.sinistro, f);
    preordine(a.destro, f);
  }
```

A parole:

1. l'interfaccia `FunzioneVoid` garantisce la presenza di un metodo `azione(Object o)`
2. il metodo `preordine` riceve un oggetto `f` di una classe che implementa l'interfaccia `FunzioneVoid`
3. sull'oggetto `f` si può invocare il metodo `f.azione(a.info)`

Implementazioni di `FunzioneVoid`

Posso passare a `preordine` solo oggetti di classi che implementano l'iterfaccia `FunzioneVoid`

Dato che le due possibili operazioni che voglio poter eseguire sui nodi sono i metodi `azione` di `Funzione4` e `FunzioneString`, queste due classi devono implementare l'interfaccia `FunzioneVoid`

```
class Funzione4 implements FunzioneVoid {
  public void azione(Object o) {
    Integer i=(Integer) o;
    if(i.intValue()<=4)
      System.out.print(i.intValue()+" ");
  }
}
```

```
class FunzioneString implements FunzioneVoid {
  static void azione(Object o) {
    if(o instanceof String)
      System.out.println(o);
  }
}
```

Riassunto

Le varie classi/metodi/interfacce:

`FunzioneVoid`
: su una variabile `f` di questo tipo di può fare `f.azione(o)`

`Funzione4`
: una classe che implementa l'interfaccia `FunzioneVoid`;
: il suo metodo `azione(o)` stampa l'oggetto passato se è un intero minore o uguale a quattro;

`FunzioneString`
: altra classe che implementa l'interfaccia `FunzioneVoid`;
: il suo metodo `azione(o)` stampa l'oggetto se è una stringa

metodo `preordine`

- prende un oggetto che si può mettere in una variabile `FunzioneVoid`
  (un oggetto di una classe che sicuramente contiene il metodo `azione`)
- per ogni nodo, invoca `f.azione(a.info)`

Generalità del meccanismo

Non è limitato agli alberi:

1. una interfaccia garantisce l'esistenza di un metodo `azione`
2. una o più classi implementano l'interfaccia
   definendo il corpo di `azione` in modo diverso
3. un metodo riceve un oggetto "di tipo interfaccia"
   (ossia, di una di queste classi)
   e invoca `azione` su di esso

Quando si invoca il metodo con un oggetto di una delle classi, viene eseguito il suo metodo `azione`

Eseguire nuove operazioni

Si possono sempre definire nuove azioni da fare sui nodi dell'albero

Esempio: stampare la versione stringa degli oggetti, ma al massimo dieci caratteri:

```
class FunzioneDieci implements FunzioneVoid {
  public void azione(Object o) {
    String s=o.toString();
    if(s.length()<10)
      System.out.println(s);
    else
      System.out.println(s.substring(0, 10));
  }
}
```

Se faccio `preordine(a, new FunzioneString())`, vengono stampate solo le stringhe

Facendo `preordine(a, new Funzione4())`, vengono stampati gli interi minori di quattro

Fancendo `preordine(a, new FunzioneDieci())`, vengono stampate le stringhe corrispondenti agli oggetti, ma al massimo dieci caratteri

Conclusione: con lo stesso metodo `preordine`, si può effettuare una operazione arbitraria su tutti i campi info dell'albero

---

Parametri del metodo `azione`

La classe `Funzione4` specifica un metodo `azione` che: "stampa gli interi minori di quattro"

Si può pensare a una variante in cui il metodo `azione` fa: "stampa gli interi minori di un certo valore"

Soluzione ovvia (sbagliata): aggiungere argomenti al metodo `azione`

Se lo faccio, devo modificare l'interfaccia `FunzioneVoid`, e poi anche il metodo `preordine` e poi anche tutte le classi che implementano `FunzioneVoid`!

Soluzione: memorizzo i parametri nell'oggetto

```
class FunzioneLimite implements FunzioneVoid {
  int limite;

  FunzioneLimite(int l) {
    limite=l;
  }

  public void azione(Object o) {
    Integer i=(Integer) o;
    if(i.intValue()<limite)
      System.out.print(i.intValue()+" ");
  }
}
```

L'invocazione sarà fatta nel seguente modo:

```
preordine(a, new FunzioneLimite(4));
```

Sequenza di operazioni

Il valore del massimo da stampare è 4.

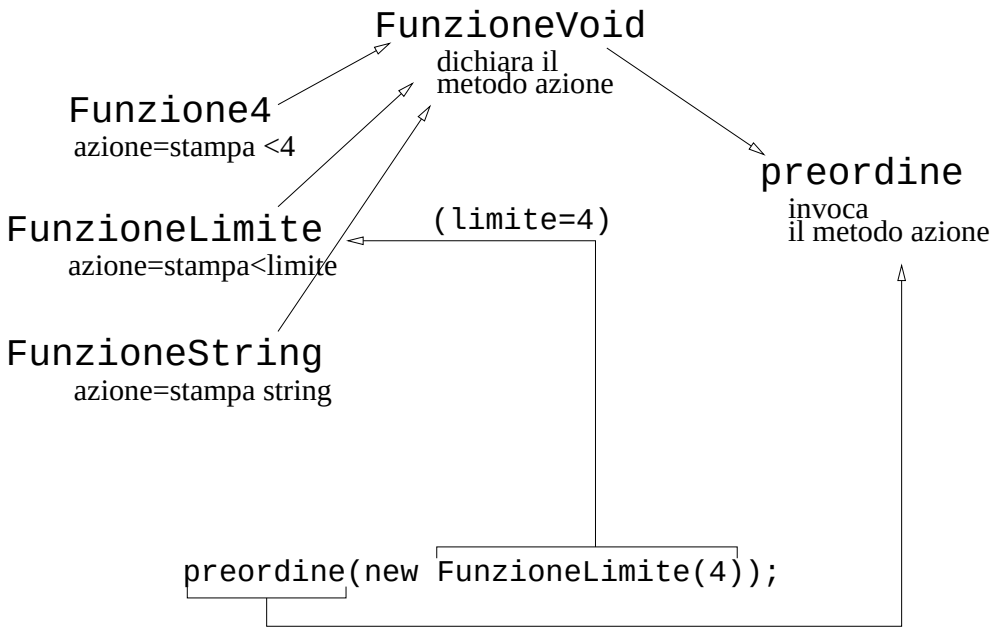
`new FunzioneLimite(4)`
viene creato un oggetto `FunzioneLimite`;
nella sua componente `limite` viene inserito dal costruttore il valore `4`

`preordine(a, new FunzioneLimite(4));`
al metodo `preordine` viene passato questo oggetto, e messo nel parametro formale `f`

`f.azione(a.info)`
questo metodo stampa tutti gli interi di valore minore di `f.limite`, che è quattro

Dichiarazioni e dati

Graficamente:



Quando si esegue `preordine(new FunzioneLimite(4));`

1. il 4 viene messo nel campo `limite` dell'oggetto `FunzioneLimite` creato
2. questo oggetto è di una classe che implementa `FunzioneVoid`
3. si può quindi passare a `preordine`
4. questa funzione invoca `oggetto.azione(a.info)` sui nodi
5. vengono stampati gli interi minori di quattro

La scelta dell'oggetto da passare a `preordine` determina il metodo da eseguire sui nodi (il metodo `azione` di `Funzione4`, oppure il metodo `azione` di `FunzioneLimite`, ecc.

Nel casi di `FunzioneLimite`, i valori che vengono messi nelle componenti vengono poi usati dal metodo

---

Come fare cose diverse sui nodi

stampare interi minori di dieci:

```
preordine(new FunzioneLimite(10));
```

stampare solo le stringhe:

```
preordine(new FunzioneStringa());
```

Il meccanismo permette di usare diversi parametri con lo stesso metodo da invocare su tutti i nodi, oppure usare diversi un diverso metodo da invocare su tutti i nodi

---

Metodi con valori di ritorno

Esempio: somma dei nodi

Soluzioni:

1. accumulo i valori trovati in una variabile d'istanza della classe
2. definisco una interfaccia in cui il metodo è `int azione(Object o, int a, int b)`

Nel secondo caso, il metodo verrà invocato con i risultati delle invocazioni ricorsive nei parametri

---

Visite con stack e code

Le pile e le code sono sequenze di oggetti con i seguenti vincoli:

1. si puo' inserire solo in coda
2. si puo' trovare/eliminare solo l'elemento in coda (stack) o in testa (coda)

Realizzano l'accesso FIFO (first-in first-out) e LIFO (last-in first-out)

Sono sempre sequenze di oggetti: si tratta più che altro di usare solo specifiche operazioni di accesso (es. inserire solo in coda)

Visite in profondità

Si ralizzano usando uno stack:

Finchè lo stack non è vuoto:

1. estrai il primo elemento
2. se è un dato, stampalo
3. se è un albero, inserisci nello stack il campo `info` e i due sottoalberi

---

Stack: principio di funzionamento

A ogni passo, si prende un elemento dallo stack

Nello stack, ci metto ora gli elementi che vanno guardati subito dopo questo

Guardo un info: lo stampo e basta

Guardo un albero: va prima stampato l'info, poi sinistro e poi destro

Gli elementi vanno messi in ordine inverso, dato che l'ultimo elemento che metto nella pila sarà il primo ad uscire

```
static void preconstack(Albero a) {
  LinkedList l=new LinkedList();
  l.add(a);

  while(l.size()>0) {
```

```
      Object o=l.getLast();
      l.removeLast();

      if(o==null)
        continue;

      if(!(o instanceof Albero)) {
        System.out.print(o+" ");
        continue;
      }

      a=(Albero) o;
      if(a!=null) {
        l.add(a.destro);
        l.add(a.sinistro);
        l.add(a.info);
      }
    }
  }
```

Visita in ampiezza

Politica differente: gli elementi li metto in una coda

L’idea è che, quando visito un nodo, metto in coda il suo info e i due sottoalberi

È quindi garantito che verranno estratti i due figli consecutivamente

```
  static void livelli(Albero a) {
    LinkedList l=new LinkedList();
    l.add(a);

    while(l.size()>0) {
      Object o=l.get(0);
      l.remove(0);

      if(o==null)
        continue;

      if(!(o instanceof Albero)) {
        System.out.print(o+" ");
        continue;
      }

      a=(Albero) o;
      if(a!=null) {
        l.add(a.info);
        l.add(a.sinistro);
        l.add(a.destro);
      }
    }
  }
```

Incapsulamento

La classe `Albero` ha il problema di permettere accesso libero alle componenti

Versione incapsulata

```
class Albero {
  private class Nodo {
    int info;
    Albero sinistro;
    Albero destro;

    public boolean equals(Object o) {
      if(o==null)
        return false;

      if(o.getClass()!=this.getClass())
        return false;

      Nodo a=(Nodo) o;

      // i campi sinistro e destro non possono
      // valere null
      return (a.info==info)&&
             (a.sinistro==this.sinistro)&&
             (a.destro==this.destro);
    }
  }

  private Nodo n;

  public Albero() {
    // n=null;
  }

  public Albero(int info, Albero sinistro, Albero destro) {
    n=new Nodo();

    n.info=info;
    n.sinistro=sinistro;
    n.destro=destro;
  }

  public boolean eVuoto() {
    return n==null;
  }

  public int getDato() {
    return n.info;
  }

  public Albero getSinistro() {
    return n.sinistro;
  }

  public Albero getDestro() {
    return n.destro;
  }
```

Uso della classe

Posso ora accedere all’albero solo attraverso i metodi

Esempio: visita in preordine

```
static void preordine(Albero a) {
  if(a.eVuoto())
    return;

  System.out.print(a.getDato()+" ");
  preordine(a.getSinistro());
  preordine(a.getDestro());
}
```

Test sugli alberi

Metodo di generazione di un albero casuale:

```
static Albero alberoRandom(int livelli) {
  Albero s, d;

  if(livelli==0)
    return null;

  if(Math.random()<0.2)
    return null;

  s=alberoRandom(livelli-1);
  d=alberoRandom(livelli-1);

  return new Albero(
    new Integer((int) (Math.random()*41-20)),
    s, d);
}
```

Stampa di un albero

Questa funzione stampa un albero

Non è necessario sapere come funziona

```
static void stampaAlbero(Albero a, String before, String after) {
  if(a==null)
    return;

  stampaAlbero(a.destro, before+"     ", before+"    |");

  System.out.print(before.substring(0,before.length()-1));

  System.out.println("+--["+a.info+"]");

  stampaAlbero(a.sinistro, after+"    |", after+"     ");
}

static void stampaAlbero(Albero a) {
  if(a==null)
```

```
    System.out.println("(albero vuoto)");
  else
    stampaAlbero(a, " ", " ");
}
```