**Dipartimento di Informatica e Sistemistica**
**Antonio Ruberti**

**“Sapienza” Università di Roma**

# Esercitazione 8

## *Corso di Tecniche di programmazione*

### *Laurea in Ingegneria Informatica*

***(Canale di Ingegneria delle Reti e dei Sistemi Informatici - Polo di Rieti)***

Anno Accademico 2007/2008

**Tutor: Ing. Diego Rughetti**

# Visita in profondità di un albero binario (1)

Usando il tipo astratto *Pila* è possibile realizzare una implementazione iterativa della visita in profondità. In particolare, presentiamo un metodo per la classe AlberoBin che realizza la visita in profondità in preordine dell'albero binario su cui è invocato.

```
public void visitaPreordineIterativa() {
        Pila p = new Pila(); // crea pila vuota p
        p.push(this); // aggiunge a p albero iniziale
        while(!p.estVuota()) {
                AlberoBin t = (AlberoBin)p.top(); // preleva elem in cima a p
                p.pop(); // elimina elem in cima a p
                if (!t.estVuota()) {
                        System.out.println(t.radice()); // stampa radice corrente
                        p.push(t.destro()); // aggiunge a p sottoalbero destro
                        p.push(t.sinistro()); // aggiunge a p sottoalbero sinistro
                }
        }
}
```

# Visita in profondità di un albero binario (2)

Osserviamo che:

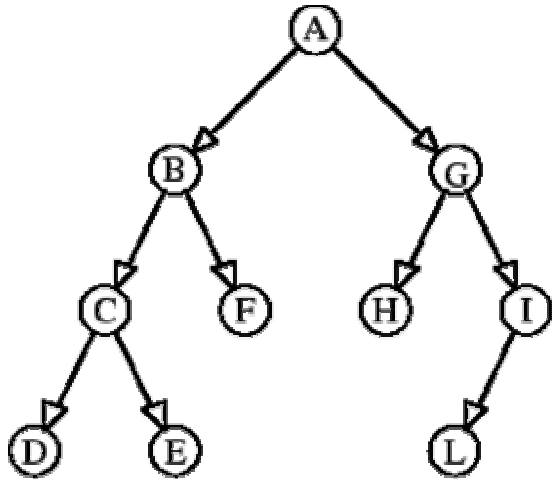
- il riferimento all'albero da visitare viene messo in cima alla pila p, inizialmente vuota, mediante l'istruzione p.push(this) prima di iniziare la visita;

- la visita vera e propria viene effettuata mediante un ciclo che ad ogni iterazione estrae un riferimento a un (sotto)albero t da visitare, stampa l'informazione associata alla radice di t, e aggiunge alla pila i riferimenti ai sottoalberi destro e sinistro di t (se t non è vuoto). In questo modo, all'iterazione successiva verrà estratto il riferimento al sottoalbero sinistro di t (che sarà in cima alla pila) e la visita proseguirà da quel nodo;

- la visita termina quando la pila si svuota, e questo avviene quando tutti i sottoalberi sono stati visitati.

# Visita in ampiezza di un albero binario (1)

Vediamo ora un altro tipo di visita chiamata **visita in ampiezza** (o **per livelli**), realizzabile in modo iterativo mediante una coda. L'idea della visita in ampiezza è quella di visitare dapprima la radice dell'albero, poi i figli della radice, poi i figli dei figli della radice, ecc. In questo modo i nodi a livello *i* saranno visitati solo dopo che tutti i nodi del livello *i* - 1 sono stati visitati.
*Esempio:* La visita in ampiezza dell'albero binario della figura seguente è: A B G C F H I D E L.




Presentiamo ora un metodo per la classe AlberoBin che realizza la visita in ampiezza dell'albero binario su cui è invocato. Si noti che esso è del tutto simile al metodo visitaPreordineIterativa, salvo che viene usata una coda invece di una pila: la diversa disciplina di accesso agli elementi dei due tipi astratti *Pila* e *Coda* rende diverso l'ordine con cui vengono elaborati i nodi dell'albero binario, e permette quindi di realizzare i due tipi di visite (in profondità e in ampiezza).

# Visita in ampiezza di un albero binario (2)

```
public void visitaAmpiezza() {
        Coda c = new Coda(); // crea coda vuota c
        c.inCoda(this); // accoda in c albero iniziale
        while(!c.estVuota()) {
                AlberoBin t = (AlberoBin)c.primo(); // preleva primo elem da c
                c.outCoda(); // elimina primo elem da c
                if (!t.estVuota()) {
                        System.out.println(t.radice()); // stampa radice corrente
                        c.inCoda(t.sinistro()); // accoda in c il sottoalbero sin
                        c.inCoda(t.destro()); // accoda in c il sottoalbero des
                }
        }
}
```

# Visita in ampiezza di un albero binario (3)

Osserviamo che:

- il riferimento all'albero da visitare viene messo nella coda c, inizialmente vuota, mediante l'istruzione c.inCoda(this) prima di iniziare la visita;

- la visita vera e propria viene effettuata mediante un ciclo che ad ogni iterazione estrae dalla coda un riferimento t a un (sotto)albero da visitare, stampa l'informazione associata alla radice di t, e aggiunge alla coda i riferimenti ai sottoalberi sinistro e destro di t, nell'ordine (se t non è vuoto). La politica FIFO della coda fa in modo che le radici di questi sottoalberi verranno visitate solo dopo che tutti i nodi allo stesso livello della radice di t saranno stati visitati.

- la visita termina quando la coda si svuota, e questo avviene quando tutti i sottoalberi sono stati visitati.

# Esercizio

Realizzare un insieme di classi java per la modellazione di due tipologie di alberi binari, uno contenente informazioni sotto forma di numeri interi, l'altro contenente informazioni sotto forma di stringhe. Dopo di che, realizzare i seguenti metodi per la manipolazione e l'utilizzo dell'opportuna tipologia di albero:

1. Metodo statico pubblico che, dati un intero *n* ed il riferimento alla radice di un albero binario *alb* i cui nodi contengono interi, restituisca il numero di occorrenze di interi minori di *n* in *alb*.
2. Metodo statico pubblico che, dato il riferimento alla radice di un albero binario *alb* i cui nodi contengono reali negativi, restituisca il minimo valore reale in *alb* (oppure 0 se l'albero è vuoto).
3. Metodo statico pubblico che, dati una stringa *s* ed il riferimento alla radice di un albero binario *alb* i cui nodi contengono stringhe, restituisca true se *s* compare in *alb*, false altrimenti.
4. Metodo statico pubblico che, dati una stringa *s* ed il riferimento alla radice di un albero binario *alb* i cui nodi contengono stringhe, restituisca il numero di occorrenze di *s* in *alb*.
5. Metodo statico pubblico che, dato il riferimento alla radice di un albero binario *alb* i cui nodi contengono interi, restituisca il numero di foglie la cui informazione sia un numero positivo (maggiore di 0).
6. Metodo statico pubblico che, dato il riferimento alla radice di un albero binario *alb*, restituisca il numero di nodi che hanno esattamente un figlio diverso dall'albero vuoto.

Facendo uso della rappresentazione collegata di alberi binari e dei tipi di dato Lista, Pila e Coda, si realizzino i metodi statici indicati nella slide successiva. In ogni metodo statico utilizzare uno solo tra i tre tipi di dato appena citati. Si considerino già realizzate le classi java che implementano i tre tipi di dato. Le interfacce di tali classi sono le seguenti (dove <TYPE> può essere sostituito dal tipo di interesse per il metodo).

```
class Lista {
        public Lista()
        public boolean empty()
        public <TYPE> firstElement()
        public Lista insertFirstElement(<TYPE> x)
        public Lista removeFirstElement()
}

class Pila {
        public Pila()
        public boolean empty()
        public <TYPE> top()
        public void push(<TYPE> x)
        public void pop()
}

class Coda {
        public Coda()
        public boolean empty()
        public <TYPE> primo()
        public void inCoda(<TYPE> x)
        public void outCoda()
}
```

1. Metodo statico pubblico che, dato il riferimento alla radice di un albero binario di ricerca *alb* i cui nodi contengono interi, restituisca la lista ordinata degli elementi dell'albero.

2. Metodo statico pubblico che, dato il riferimento alla radice di un albero binario *alb* i cui nodi contengono Stringhe, restituisca una coda contenente le informazioni delle foglie ordinate da sinistra a destra. Il primo elemento della coda risultante dovrà essere la foglia estrema a sinistra.

3. Realizzare un metodo statico pubblico che, dato il riferimento alla radice di un albero binario *alb* i cui nodi contengono interi, restituisca la Lista dei padri di esattamente due foglie in qualunque ordine.

4. Realizzare un metodo statico pubblico che, dato il riferimento alla radice di un albero binario *alb* i cui nodi contengono Stringhe, restituisca una pila contenente tutte le informazioni nei nodi, in modo tale che svuotandola si ottenga l'effetto equivalente ad una visita in preordine

5. Realizzare un metodo statico pubblico che, dato il riferimento alla radice di un albero binario *alb* i cui nodi contengono Stringhe, restituisca una coda contenente tutte le informazioni nei nodi, in modo tale che svuotandola si ottenga l'effetto equivalente ad una visita simmetrica.

# Soluzione

```
public class NodoBinarioInteri{
  public int info;
  public NodoBin sinistro;
  public NodoBin destro;
  public NodoBinarioInteri(){}
}

public class AlberoBinarioInteri{
        public NodoBinarioInteri radice;
        public AlberoBinarioInteri(){
                this.radice = null;
        }
}
```

```
public class AlberoBinarioStringhe{
        public NodoBinarioStringhe radice;
        public AlberoBinarioStringhe(){
                this.radice = null;
        }
}

public class NodoBinarioStringhe{
        public String info;
        public NodoBin sinistro;
        public NodoBin destro;
        public NodoBinarioStringhe(){}
}
```

```
public static int OccorrenzeMinori(NodoBinarioInteri r, int n){
        if (a == null) //albero vuoto
                return 0;
        else {
                int i = OccorrenzeMinori(a.sinistro);    //sottoalb. sin.
                int j = OccorrenzeMinori(a.destro);      //sottoalb. des.
                if(r.info < n){
                        return (i + j + 1);
                }else{
                        return (i + j);
                }
        }
}
public static int Minore(NodoBinarioInteri r, int n){
        if (a == null)
                return 0;
        else {
                int i = Minore(a.sinistro);
                int j = Minore(a.destro);
                if(r.info < i && r.info < j){
                        return r.info;
                }else if(i<j){
                        return i;
                }else{
                        return j;
                }
        }
}
```

```
public static boolean presente(NodoBinarioStringhe a, String x) {
		if (a == null)
			return false;
		else if (a.info.equals(x))
			return true;
		else
			return presente(a.sinistro,x) || presente(a.destro,x);
}

public static int contaFogliePositive(NodoBinarioInteri a) {
		if (a == null)
			return 0;
		else if (a.sinistro == null && a.destro == null && a.info > 0) // foglia
			return 1;
		else {
			int fogliePostiveSx = contaFogliePositive(a.sinistro);   //visita sottoalb. sin.
			int fogliePositiveDx = contaFogliePositive(a.destro);   //visita sottoalb. des.
			return fogliePositiveSx + fogliePositiveDx;                //opera su nodo corrente
		}
}
```

```
public static int contaNodiSoloFiglio(NodoBinarioInteri a) {
        if (a == null)
                return 0;
        else if (a.sinistro == null && a.destro == null)
                return 0;
        else {
                int sx = contaNodiSoloFiglio(a.sinistro); //visita sottoalb. sin.
                int dx = contaNodiSoloFiglio(a.destro);   //visita sottoalb. des.
                if(a.sinistro == null || a.destro == null)
                        return (sx + dx + 1);
                else
                        return (sx + dx);
        }
}


public static int OccorrenzeStringa(NodoBinarioStringa r, String s){
        if (a == null)
                return 0;
        else {
                int i = OccorrenzeStringa(a.sinistro);
                int j = OccorrenzeStringa(a.destro);
                if(r.info.equals(s)){
                        return (i + j + 1);
                }else{
                        return (i + j);
                }
        }
}
```

```
public static Lista getListaOrdinata(NodoBinarioInteri r){
        Lista l = new Lista();
        doOrderedList(r, l);
        return l;
}

private static void doOrderedList(NodoBinarioInteri r, Lista l){
        if (a == null)
                return;
        else{
                doOrderedList(r.destra, l);
                l.insertFirstElement(r.info);
                doOrderedList(r.sinistra, l);
        }
}
```

```
private static Coda getCodaFoglie(NodoBinarioStringa r){
        Coda c = new Coda();
        doOrderedCoda(r, c);
        return c;
}

private static void doOrderedCoda(NodoBinarioStringa r, Coda c){
        if (r == null)
                return;
        else{
                doOrderedCoda(r.sinistra, c);
                doOrderedCoda(r.destra, c);
                if(r.sinistra == null && r.destra == null)
                c.inCoda(r.info);
        }
}
```

```
private static Coda getListaPadriFoglie(NodoBinarioInteri r){
        Lista c = new Lista();
        doOrderedCoda(r, c);
        return c;
}

private static boolean doPadriLista(NodoBinarioInteri r, Lista l){
        if (r == null)
                return false;
        else{
                boolean a = doPadriLista(r.sinistra, c);
                boolean b = doPadriLista(r.destra, c);
                if(a && b)
                        l.insertFirstElement(r.info);
                if(r.sinistra == null && r.destra == null)
                        return true;
                else
                        return false;
        }
}
```

```
//RIPASSO

public static void visitaPreordine(NodoBin a) {
        if (a == null) //albero vuoto
                return;
        else {
                System.out.print(a.info + " ");  //opera su nodo corrente
                visitaPreordine(a.sinistro);     //visita sottoalb. sin.
                visitaPreordine(a.destro);       //visita sottoalb. des.
        }
}

private static Pila getPilaPreordine(NodoBinarioStringa r){
        Pila p = new Pila();
        doPilaPreordine(r, p);
}

private static void doPilaPreordine(NodoBinarioStringa r, Pila p){
        if(r == null)
                return;
        else{
                doPilaPreordine(r.destra, p);
                doPilaPreordine(r.sinistra, p);
                p.push(r.info);
        }
}
```

```
//RIPASSO

public static void visitaSimmetrica(NodoBin a) {
		if (a == null) //albero vuoto
				return;
		else {
				visitaSimmetrica(a.sinistro);    //visita sottoalb. sin.
				System.out.print(a.info + " ");  //opera su nodo corrente
				visitaSimmetrica(a.destro);      //visita sottoalb. des.
		}
}

private static Coda getCodaSimmetrica(NodoBinarioStringa r){
		Coda c = new Coda();
		doCodaSimmetrica(r, c);
}

private static void doCodaSimmetrica(NodoBinarioStringa r, Coda c){
		if(r == null)
				return;
		else{
				doCodaSimmetrica(r.sinistra, p);
				p.push(r.info);
				doCodaSimmetrica(r.destra, p);
		}
}
```