Azzolini Riccardo 2019-03-25

Visita iterativa di alberi binari

1 Eliminazione della ricorsione nella visita degli alberi

La visita (in ordine anticipato, posticipato o simmetrico) degli alberi binari è implementata mediante due chiamate ricorsive, almeno una delle quali non è in coda.

Per eliminare totalmente la ricorsione, è quindi necessario utilizzare una pila.

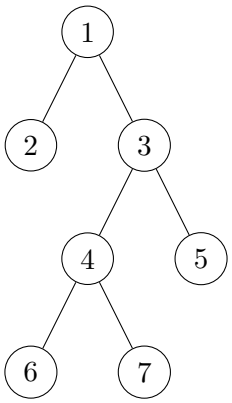
2 Visita iterativa in ordine anticipato

```
private void preOrderIt(Node r) {
    Stack<Node> s = new Stack<Node>();
    if (r != null) s.push(r);

    while (!s.isEmpty()) {
        r = s.top(); s.pop();
        r.visit();
        if (r.dx != null) { s.push(r.dx); }
        if (r.sx != null) { s.push(r.sx); }
    }
}
```

Osservazione: Il figlio destro di ogni nodo deve essere salvato nello stack *prima* di quello sinistro: in questo modo il figlio sinistro rimane in cima e così il sottoalbero sinistro viene (correttamente) visitato prima di quello destro.

2.1 Esempio



Pila `s`	Visitato
`1`	
`3,2`	1
`3`	2
`5,4`	3
`5,7,6`	4
`5,7`	6
`5`	7
	5

3 Visita iterativa in ordine simmetrico

1. Si elimina la seconda chiamata ricorsiva, che è in coda:

```
private void inOrderIt(Node r) {
    while (r != null) {
        inOrderIt(r.sx);
        r.visit();
        r = r.dx;
    }
}
```

2. Si elimina l'altra chiamata ricorsiva usando uno stack:

```
private void inOrderIt(Node r) {
    Stack<Node> s = new Stack<Node>();
    if (r != null) s.push(r);

    while (!s.isEmpty()) {
```

```
            // (stato 1)

            r = s.top(); s.pop();

            // (stato 2)

            while (r != null) {
                s.push(r);
                r = r.sx;
            }

            // (stato 3)

            if (!s.isEmpty()) {
                r = s.top(); s.pop();
                r.visit();
                s.push(r.dx);
            }
        }
    }
```

Funzionamento: Per prima cosa, viene salvata sullo stack la radice. Poi, finché lo stack non è vuoto:

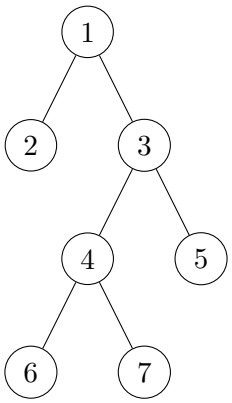
1. Viene prelevato il nodo in cima allo stack.
2. Dal nodo prelevato, si continua a passare al figlio sinistro, finché non si raggiunge un riferimento nullo. Il cammino percorso (compreso il nodo di partenza, che era stato prelevato al punto 1, ma non il riferimento nullo finale) viene salvato sullo stack.
3. Si ritorna all'ultimo nodo del cammino (che non ha figlio sinistro). Questo viene visitato, poi si salva sullo stack il suo figlio destro, *anche se è nullo*.
4. All'iterazione successiva del ciclo `while (!s.isEmpty())` (esterno):
    - se il figlio destro non era nullo, l'intero procedimento si ripete a partire da esso, in modo da visitare tale nodo e il sottoalbero di cui è radice;
    - altrimenti, il riferimento nullo presente in cima allo stack fa saltare il ciclo `while (r != null)` (interno), poi si estrae dallo stack il nodo precedente del cammino, che viene visitato, seguito dal suo sottoalbero destro.

In altre parole, per ogni sottoalbero l'algoritmo:

1. percorre il cammino di figli sinistri, fino ad arrivare a un nodo che non ha figlio sinistro;

2. risale lungo il cammino, visitando ciascun nodo e il suo sottoalbero destro.

3.1 Esempio



Stato	`r`	Pila `s`	Visitato
1	`1`	`1`	
2	`1`		
3	`null`	`1,2`	
1	`2`	`1,null`	2
2	`null`	`1`	
3	`null`	`1`	
1	`1`	`3`	1
2	`3`		
3	`null`	`3,4,6`	
1	`6`	`3,4,null`	6
2	`null`	`3,4`	
3	`null`	`3,4`	
1	`4`	`3,7`	4
2	`7`	`3`	
3	`null`	`3,7`	
1	`7`	`3,null`	7
2	`null`	`3`	
3	`null`	`3`	
1	`3`	`5`	3
2	`5`		
3	`null`	`5`	
1	`5`	`null`	5
2	`null`		
3	`null`		
fine			

4 Visita iterativa in ordine posticipato

Ogni nodo viene immesso due volte nello stack: una volta per prenotare la visita di tutto il suo sottoalbero, e una per prenotare la visita del singolo nodo.

Si usano quindi due stack paralleli: uno di nodi, `sn`, e uno di booleani, `sb`.[1] Per ogni nodo in `sn`, lo stack `sb` contiene:

- `true` se deve essere visitato l'intero sottoalbero;
- `false` se si deve visitare solo il singolo nodo.

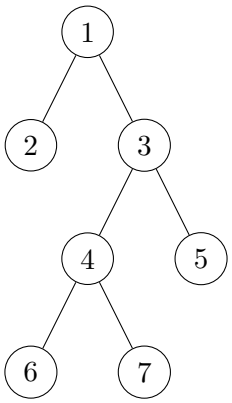
```
private void postOrderIt(Node r) {
    Stack<Node> sn = new Stack<Node>();
    Stack<Boolean> sb = new Stack<Boolean>();
    if (r != null) {
        sn.push(r); sb.push(true);
    }

    while (!sn.isEmpty()) {
        r = sn.top(); sn.pop();
        boolean f = sb.top(); sb.pop();

        if (f) { // Visita il sottoalbero
            sn.push(r); sb.push(false);
            if (r.dx != null) {
                sn.push(r.dx); sb.push(true);
            }
            if (r.sx != null) {
                sn.push(r.sx); sb.push(true);
            }
        } else { // Visita solo il nodo
            r.visit();
        }
    }
}
```

[1] In alternativa, si potrebbe usare un singolo stack di record, ciascuno contenente un nodo e un booleano.

4.1 Esempio



Pile `sn` e `sb`	Visitato
`1T`	
`1F,3T,2T`	
`1F,3T,2F`	
`1F,3T`	2
`1F,3F,5T,4T`	
`1F,3F,5T,4F,7T,6T`	
`1F,3F,5T,4F,7T,6F`	
`1F,3F,5T,4F,7T`	6
`1F,3F,5T,4F,7F`	
`1F,3F,5T,4F`	7
`1F,3F,5T`	4
`1F,3F,5F`	
`1F,3F`	5
`1F`	3
	1

5 Complessità

Per un albero con n nodi, tutte queste procedure iterative hanno complessità (in base al CCU) uguale a quella delle versioni ricorsive:

- tempo $\Theta(n)$, perché ogni nodo transita dallo stack un numero costante di volte (1 per l'ordine anticipato, 2 per gli altri), e ogni volta si esegue un numero costante di operazioni a costo costante (così come per ogni nodo viene eseguita una sola chiamata ricorsiva);
- spazio $O(n)$, e in particolare
  - $O(1) = \Theta(1)$ nel caso migliore

– $\Theta(n)$ nel caso peggiore

perché il numero di nodi contenuti nello stack, e quindi la memoria occupata, è asintoticamente uguale al numero di record di attivazione per le chiamate ricorsive.