Lezione 7 – Alberi binari: visite e alberi di ricerca

Informatica

6 Maggio 2015

Operazioni sugli alberi: visita

La *visita* di un albero esplora tutti i nodi ed esegue una qualche operazione su di essi. Per esempio, ne stampa l'etichetta:

1. stampa l'etichetta della radice e poi stampa le *etichette del sottoalbero sinistro* seguite da quelle del *sottoalbero destro*
2. la visita si implementa con una funzione *ricorsiva*

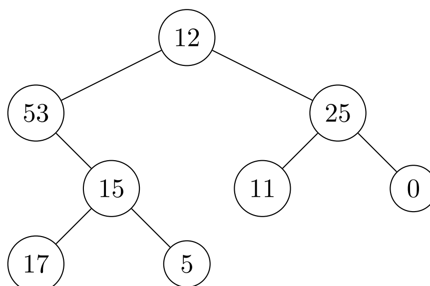
Questo tipo di visita è detto *prefissa*. Ne esistono altri due tipi:

- visita *infissa*: stampa prima il sottoalbero sinistro, poi la radice e infine il sottoalbero destro
- visita *postfissa*: stampa il sottoalbero sinistro, poi quello destro ed infine la radice

```
def prefix_visit(t):
    if not is_empty(t):
        print(label(t))
        prefix_visit(first_child(t))
        prefix_visit(second_child(t))

def infix_visit(t):
    if not is_empty(t):
        infix_visit(first_child(t))
        print(label(t))
        infix_visit(second_child(t))

def postfix_visit(t):
    if not is_empty(t):
        postfix_visit(first_child(t))
        postfix_visit(second_child(t))
        print(label(t))
```

Visita: esempi

Dato l'albero binario



qual’è il risultato delle tre visite?

```
t = bin(12,
        bin(53, tree(), bin(15, leaf(17), leaf(5))),
        bin(25, leaf(11), leaf(0)))
prefix_visit(t)
infix_visit(t)
postfix_visit(t)
```

La visita prefissa produce

```
12    53    15    17     5    25    11     0
```

La visita infissa produce

```
53    17    15     5    12    11    25     0
```

La visita postfissa produce

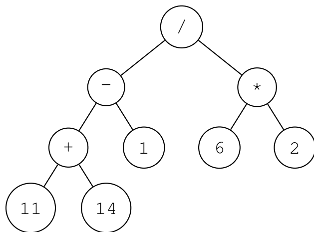
```
17     5    15    53    11     0    25    12
```

Esercizio: alberi ed espressioni

Con gli alberi binari si rappresentano *espressioni aritmetiche*:

- I nodi interni sono etichettati con gli operatori `+, -, *, /`
- Le foglie sono etichettate con operandi di tipo `int`

Esempio: `((11 + 14) - 1) / (6 * 2)`



- la radice è etichettata con *l’operatore principale*
- i sottoalberi rappresentano le due *sottoespressioni*

Scrivere una funzione che stampa l’espressione e una che valuta l’espressione.

```
def expr(t):
    if not is_empty(t):
        a = label(t)
        if type(a) == int:
            return a
        elif a == '+':
            return expr(first_child(t)) + expr(second_child(t))
        elif a == '-':
            return expr(first_child(t)) - expr(second_child(t))
        elif a == '*':
            return expr(first_child(t)) * expr(second_child(t))
        elif a == '/':
            return expr(first_child(t)) // expr(second_child(t))
    return None
```

```
def print_expr(t):
    print(print_expr_aux(t))

def print_expr_aux(t):
    if not is_empty(t):
        a = label(t)
        if type(a) == int:
            return str(a)
        else:
            s = '(' + print_expr_aux(first_child(t)) + str(a)
            s += print_expr_aux(second_child(t)) + ')'
            return s
    return ''

print(" --- ESPRESSIONI --- ")
e = bin('/', bin('-', bin('+', leaf(11), leaf(14)), leaf(1)),
        bin('*', leaf(6), leaf(2)))
print_expr(e)
print(expr(e))
```

Alberi binari di ricerca

Un *albero binario di ricerca* è un albero binario con le seguenti proprietà:

- le etichette dei provengono da un insieme ordinato (p.es. `int`)
- se un nodo è etichettato con `x`:
  - tutte le etichette del sottoalbero sinistro sono *minori di* `x`
  - tutte le etichette del sottoalbero destro sono *maggiori di* `x`

Operazioni su un albero binario di ricerca:

Visita ordinata dall'elemento più piccolo al più grande

Ricerca di un elemento all'interno dell'albero

Inserimento di un nuovo elemento

Verifica che un albero sia un albero di ricerca

La visita ordinata si implementa con una *visita infissa* dell'albero.
Ricerca e inserimento si possono implementare come segue.

```
def search(n, t):
    if is_empty(t):
        return False
    r = label(t)
    if r == n:
        return True
    elif n < r:
        return search(n, first_child(t))
    else:
        return search(n, second_child(t))
```

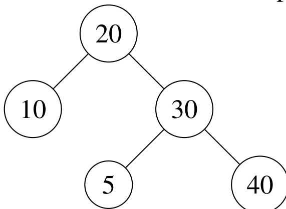
```
def insert(n, t):
    if is_empty(t):
        return leaf(n)
    r = label(t)
    if r == n:
        return t
    elif n < r:
        return bin(r, insert(n, first_child(t)), second_child(t))
    else:
        return bin(r, first_child(t), insert(n, second_child(t)))

print(" --- ALBERI DI RICERCA --- ")
n = 1
t = tree()
while n != 0:
    n = eval(input("Inserisci un numero (0 per terminare):"))
    if n != 0:
        t = insert(n, t)
print("Riepilogo dei numeri inseriti:")
infix_visit(t)
n = 1
while n != 0:
    n = eval(input("Inserisci un numero (0 per terminare):"))
    if search(n, t):
        print("Il numero è presente nell'albero")
    else:
        print("Il numero non è presente nell'albero")
```

Alberi binari di ricerca: verifica

Per verificare se un albero binario è un albero di ricerca si può pensare di procedere con una *visita*:

- se l'etichetta del nodo corrente è più piccola di quella del figlio sinistro, ritorna `False`
- se l'etichetta del nodo corrente è più grande di quella del figlio destro, ritorna `False`
- procedi ricorsivamente sul sottoalbero sinistro e destro

Domanda: cosa succede per l'albero in figura?



L'algoritmo intuitivo ritorna `True`. Ma l'albero non è un albero di ricerca, poiché il nodo 5 non rispetta la proprietà: è nel sottoalbero destro del nodo 20.

Come si risolve il problema? Per prendere una decisione non è sufficiente guardare i valori del nodo corrente e dei suoi figli. Dobbiamo tener traccia dei valori presenti nei predecessori. Nel caso sopra, quando scendiamo nel sottoalbero destro della radice dobbiamo ricordarci che i valori devono essere più grandi di 20.

```
def is_bst_aux(t, min, max):
    if is_empty(t):
        return True
    r = label(t)
    if r > max or r < min:
        return False
    if not is_bst_aux(first_child(t), min, r):
        return False
    return is_bst_aux(second_child(t), r, max)

def is_bst(t):
    return is_bst_aux(t, float("-inf"), float("inf"))

print(" --- VERIFICA ALBERI DI RICERCA --- ")
t = bin(20, leaf(10), bin(30, leaf(25), leaf(40)))
print(is_bst(t))
t = bin(20, leaf(10), bin(30, leaf(5), leaf(40)))
print(is_bst(t))
```