Laboratorio di Algoritmi e Strutture Dati
2020/2021 — Seconda parte

Mattia Bonaccorsi — 124610 – bonaccorsi.mattia@spes.uniud.it
Muhamed Kouate — 137359 – kouate.muhamed@spes.uniud.it
Enrico Stefanel — 137411 – stefanel.enrico@spes.uniud.it
Andriy Torchanyn — 139535 – torchanyn.andriy@spes.uniud.it

26 aprile 2021

Indice

1 Alberi binari di ricerca semplici

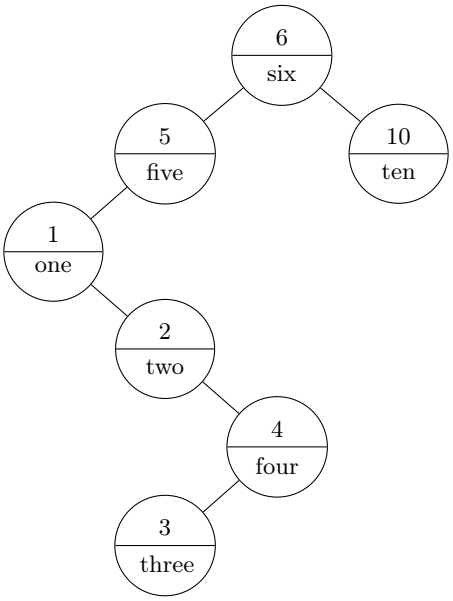
1.1 Definizione di *BST*

Un *albero binario di ricerca* (o BST) T è una struttura dati ad albero, in cui valgono le seguenti proprietà:

$$\begin{aligned} \forall x \in T,\ \forall y \in left(T) \rightarrow y.key < x.key \\ \forall x \in T,\ \forall z \in right(T) \rightarrow z.key > x.key \end{aligned} \tag{$\star$}$$

dove $k.key$ indica il valore della chiave di k, e $left(B)$ (rispettivamente $right(B)$) indica il sotto-albero sinistro (rispettivamente destro) di B.

Esempio Un *BST* di tipo semplice, in cui ogni nodo contiene una chiave numerica dell'insieme $\{1, 2, 3, 4, 5, 6, 10\}$ e un campo alfanumerico di tipo stringa, è il seguente:



Bisogna notare che non è l'unico *BST* costruibile partendo dallo stesso insieme di chiavi. Un'alternativa, per esempio, potrebbe essere stata quella di utilizzare il valore minore come chiave per la radice dell'albero, e attaccare in ordine crescente le altre chiavi, ognuna come figlio destro del nodo precedente.

1.2 Implementazione della struttura dati

```
1 class Node():
2     def __init__(self, value, str_name):
3         self.key = value
4         self.name = str_name
5         self.left = None
```

```
6           self.right = None
7
8
9   def bst_insert(root, value, str_name):
10      """
11      insert a key value in a tree
12      :param root: BSTNode object that represents
13          the root of the tree
14      :param value: an integer representing the value to insert
15      :param str_name: a string corresponding to
16          the literal format of the value
17      :return: a BSTNode object
18      """
19      if root is None:
20          return Node(value, str_name)
21
22      if value < root.key:
23          root.left = bst_insert(root.left, value, str_name)
24      else:
25          root.right = bst_insert(root.right, value, str_name)
26
27      return root
28
29
30  def bst_find(root, value):
31      """
32      print the found value in a literal format
33      :param root: BSTNode object that represents
34          the root of the tree
35      :param value: an integer representing the value to find
36      """
37      if root is None:
38          return
39
40      if root.key == value:
41          print(root.name)
42
43      if root.key < value:
44          return bst_find(root.right, value)
45
46      return bst_find(root.left, value)
47
48
49  def bst_show(root):
50      """
51      print the tree in a preorder visit
52      :param root: BSTNode object that represents
53          the root of the tree
54      """
```

```
55      if root:
56          print(
57              str(root.key), root.name,
58              sep=":",
59              end=" "
60              )
61          bst_show(root.left)
62          bst_show(root.right)
63      else:
64          print("NULL", end=" ")
```

sources/bst.py

1.3 Difetti dei *BST*

Questa struttura dati ha alcuni difetti ...

2 Alberi binari di ricerca di tipo AVL

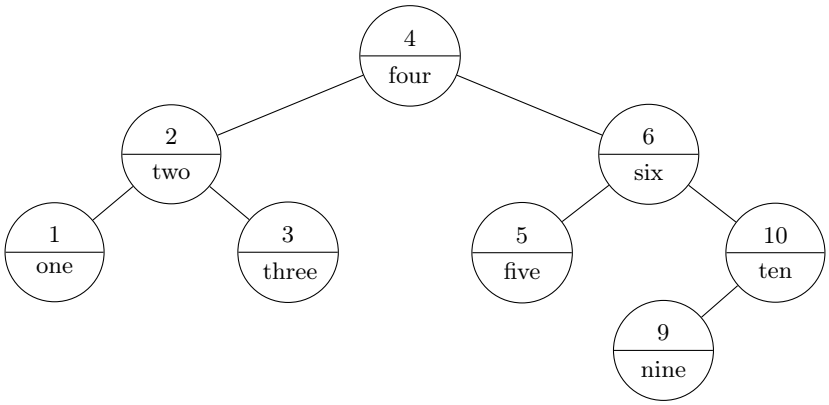
2.1 Definizione di Albero *AVL*

Un *albero AVL* T è un *BST* ($\star$), in cui vale la seguente proprietà:

$$\forall x \in T \rightarrow |h(left(x)) - h(right(x))| \leq 1 \qquad (*)$$

dove $h(k)$ indica il valore dell'altezza dell'albero radicato in k, e $left(B)$ (rispettivamente $right(B)$) indica il sotto-albero sinistro (rispettivamente destro) di B.

Esempio Un Albero *AVL* in cui ogni nodo contiene una chiave numerica dell'insieme $\{1, 2, 3, 4, 5, 6, 9, 10\}$ e un campo alfanumerico di tipo stringa, è il seguente:



, dove, ad esempio, $left(\texttt{root})$ ha altezza 2, mentre $right(\texttt{root})$ ha altezza 3.

2.2 Implementazione della struttura dati

```
class AVLNode():
    def __init__(self, value, str_name):
        self.key = value
        self.name = str_name
        self.left = None
        self.right = None
        self.height = 1


def avl_insert(root, value, str_name):
    """
    insert a key value in a tree
    :param root: AVLNode object that represents
        the root of the tree
    :param value: an integer representing the value to insert
    :param str_name: a string corresponding to
        the literal format of the value
    :return: an AVLNode object
    """
    if root is None:
        return AVLNode(value, str_name)

    if value < root.key:
        root.left = avl_insert(root.left, value, str_name)
    else:
        root.right = avl_insert(root.right, value, str_name)

    root.height=1+max(getHeight(root.left),getHeight(root.right))

    balance = getBalance(root)

    # LL
    if balance > 1 and value < root.left.key:
        return rightRotate(root)

    # RR
    if balance < -1 and value > root.right.key:
        return leftRotate(root)

    # LR
    if balance > 1 and value > root.left.key:
        root.left = leftRotate(root.left)
        return rightRotate(root)

    # RL
    if balance < -1 and value < root.right.key:
        root.right = rightRotate(root.right)
```

```
48          return leftRotate(root)
49
50      return root
51
52
53  def avl_show(root):
54      """
55      print the tree in a preorder visit
56      :param root: AVLNode object that represents
57          the root of the tree
58      """
59      if root:
60          print(
61              str(root.key), root.name, str(root.height),
62              sep=":",
63              end=" "
64              )
65          avl_show(root.left)
66          avl_show(root.right)
67      else:
68          print("NULL", end=" ")
69
70
71  def avl_find(root, value):
72      """
73      print the found value in a literal format
74      :param root: AVLNode object that represents
75          the root of the tree
76      :param value: an integer representing the value to find
77      """
78      if root is None:
79          return
80
81      if root.key == value:
82          print(root.name)
83
84      if root.key < value:
85          return avl_find(root.right, value)
86
87      return avl_find(root.left, value)
```

sources/avl.py

2.3 Miglioramenti degli Alberi AVL rispetto ai *BST*

Questa struttura dati migliora alcuni aspetti dei *BST* ...

3 Alberi binari di ricerca di tipo Red-Black

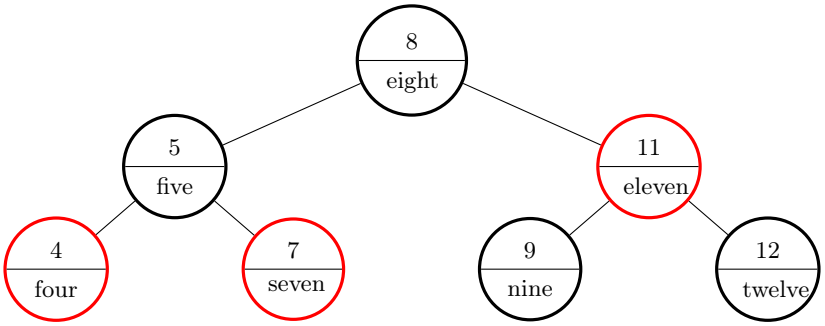
3.1 Definizione di *RB Tree*

Un *albero di tipo Red-Black* (o *RB Tree*) T è un *BST* ($\star$), in cui ogni nodo ha associato un campo "colore", che può assumere valore *rosso* o *nero*, ed inoltre vale che:

$$\forall x \in T \rightarrow h_b(left(x)) = h_b(right(x)) \qquad (\bullet)$$

dove $h_b(x)$ indica l'altezza nera dell'albero radicato in x, ovvero il massimo numero di nodi neri lungo un possibile cammino da x a una foglia.

Esempio



3.2 Implementazione della struttura dati

```
1  class RBTNode():
2      def __init__(self, value, str_name):
3          self.key = value
4          self.name = str_name
5          self.parent = None
6          self.left = None
7          self.right = None
8          self.color = "red"
9
10
11 class RedBlackTree():
12     def __init__(self):
13         self.TNIL = RBTNode(None, None)
14         self.TNIL.color = "black"
15         self.TNIL.left = None
16         self.TNIL.right = None
17         self.root = self.TNIL
18
19     def left_rotate(self, x):
20         y = x.right
21         x.right = y.left
```

```
22
23          if y.left != self.TNIL:
24              y.left.parent = x
25
26          y.parent = x.parent
27
28          if x.parent == self.TNIL:
29              self.root = y
30          elif x == x.parent.left:
31              x.parent.left = y
32          else:
33              x.parent.right = y
34
35          y.left = x
36          x.parent = y
37
38      def right_rotate(self, x):
39          y = x.left
40          x.left = y.right
41
42          if y.right != self.TNIL:
43              y.right.parent = x
44
45          y.parent = x.parent
46
47          if x.parent == self.TNIL:
48              self.root = y
49          elif x == x.parent.right:
50              x.parent.right = y
51          else:
52              x.parent.left = y
53
54          y.right = x
55          x.parent = y
56
57      def rbt_insert(self, value, str_name):
58
59          z = RBTNode(value, str_name)
60          z.left = self.TNIL
61          z.right = self.TNIL
62
63          y = self.TNIL
64          x = self.root
65
66          while x != self.TNIL:
67              y = x
68              if z.key < x.key:
69                  x = x.left
70              else:
```

```
71                  x = x.right
72
73          z.parent = y
74
75          if y == self.TNIL:
76              self.root = z
77          elif z.key < y.key:
78              y.left = z
79          else:
80              y.right = z
81
82          self.insert_fix_up(z)
83
84      def insert_fix_up(self, z):
85          while z.parent.color == "red":
86              if z.parent == z.parent.parent.right:
87                  y = z.parent.parent.left
88                  if y.color == "red":
89                      y.color = "black"
90                      z.parent.color = "black"
91                      z.parent.parent.color = "red"
92                      z = z.parent.parent
93                  else:
94                      if z == z.parent.left:
95                          z = z.parent
96                          self.right_rotate(z)
97                      z.parent.color = "black"
98                      z.parent.parent.color = "red"
99                      self.left_rotate(z.parent.parent)
100             else:
101                 y = z.parent.parent.right
102
103                 if y.color == "red":
104                     y.color = "black"
105                     z.parent.color = "black"
106                     z.parent.parent.color = "red"
107                     z = z.parent.parent
108                 else:
109                     if z == z.parent.right:
110                         z = z.parent
111                         self.left_rotate(z)
112                     z.parent.color = "black"
113                     z.parent.parent.color = "red"
114                     self.right_rotate(z.parent.parent)
115             if z == self.root:
116                 break
117         self.root.color = "black"
118
119
```

```python
def rbt_show(root):
    """
    print the tree in a preorder visit
    :param root: RBTNode object that represents
        the root of the tree (accessed by t.root in the calling)
    """
    if root.key is None:
        print("NULL", end=" ")
        return
    if root.name is None:
        print("NULL", end=" ")
        return
    if root:
        print(
            str(root.key), root.name, str(root.color),
            sep=":",
            end=" "
            )
        rbt_show(root.left)
        rbt_show(root.right)
    else:
        print("NULL", end=" ")


def rbt_find(root, value):
    """
    print the found value in a literal format
    :param root: RBTNode object that represents
        the root of the tree (accessed by t.root in the calling)
    :param value: an integer representing the value to find
    """
    if root.key is None:
        return
    if root.name is None:
        return
    if root is None:
        return

    if root.key == value:
        print(root.name)

    if root.key < value:
        return rbt_find(root.right, value)

    return rbt_find(root.left, value)
```

sources/rbt.py