

Laboratorio di Algoritmi e Strutture Dati

2020/2021 — Seconda parte

Mattia Bonaccorsi — 124610 – bonaccorsi.mattia@spes.uniud.it

Muhamed Kouate — 137359 – kouate.muhammed@spes.uniud.it

Enrico Stefanel — 137411 – stefanel.enrico@spes.uniud.it

Andriy Torchanyyn — 139535 – torchanyyn.andriy@spes.uniud.it

18 maggio 2021

Indice

1	Alberi binari di ricerca semplici	2
1.1	Definizione di <i>BST</i>	2
1.2	Implementazione della struttura dati	2
1.2.1	Osservazioni sull'implementazione della struttura dati	3
2	Alberi binari di ricerca di tipo AVL	5
2.1	Definizione di Albero <i>AVL</i>	5
2.2	Implementazione della struttura dati	5
3	Alberi binari di ricerca di tipo Red-Black	7
3.1	Definizione di <i>RB Tree</i>	7
3.2	Implementazione della struttura dati	7
4	Calcolo della complessità	10
4.1	Caso random	10
4.2	Caso sorted	10

1 Alberi binari di ricerca semplici

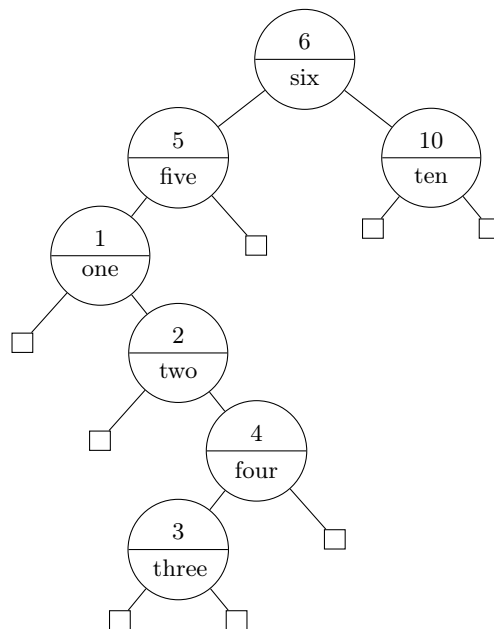
1.1 Definizione di *BST*

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

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

dove $k.\text{key}$ indica il valore della chiave di k , e $\text{left}(B)$ (rispettivamente $\text{right}(B)$) 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

Per implementare la struttura dati dell'Albero binario di ricerca semplice, abbiamo innanzitutto bisogno di definire una classe `Node` per le istanze dei Nodi che compongono il *BST*:

```

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

```

sources/Node.py

Una volta definita la classe `Node`, possiamo procedere con l'implementazione dell'inserimento di un Nodo nel BST:

```

1 def bst_insert(root, value, str_name):
2     if root is None:
3         return Node(value, str_name)
4     if value < root.key:
5         root.left = bst_insert(root.left, value, str_name)
6     else:
7         root.right = bst_insert(root.right, value, str_name)
8     return root

```

sources/bst.py

Definiamo poi una procedura, anche questa ricorsiva, per la ricerca di un Nodo all'interno di un Albero:

```

1 def bst_find(root, value):
2     if root is None:
3         return
4     if root.key == value:
5         return root.name
6     if root.key < value:
7         return bst_find(root.right, value)
8     return bst_find(root.left, value)

```

sources/bst.py

1.2.1 Osservazioni sull'implementazione della struttura dati

Le procedure per l'inserimento e la ricerca di un nodo all'interno di un BST sono state scritte in maniera ricorsiva, per chiarezza. Essendo però una *ricorsione di coda*, è immediato trasformare le funzioni per ottenere delle funzioni iterative.

La funzione per l'inserimento, scritta in maniera iterativa, sarebbe la seguente:

```

1 def bst_insert_iterative(root, value, str_name):
2     newnode = Node(value, str_name)
3     x = root
4     y = None
5     while (x != None):
6         y = x

```

```

7         if (value < x.key):
8             x = x.left
9         else:
10            x = x.right
11    if (y == None):
12        y = newnode
13    elif (value < y.key):
14        y.left = newnode
15    else:
16        y.right = newnode
17    return y

```

sources/bst.py

, mentre la funzione di ricerca sarebbe scritta in questo modo:

```

1  def bst_find_iterative(root, value):
2      x = root
3      while x is not None:
4          if x.key == value:
5              return x.name
6          if x.key < value:
7              x = x.right
8          else:
9              x = x.left
10     return

```

sources/bst.py

2 Alberi binari di ricerca di tipo AVL

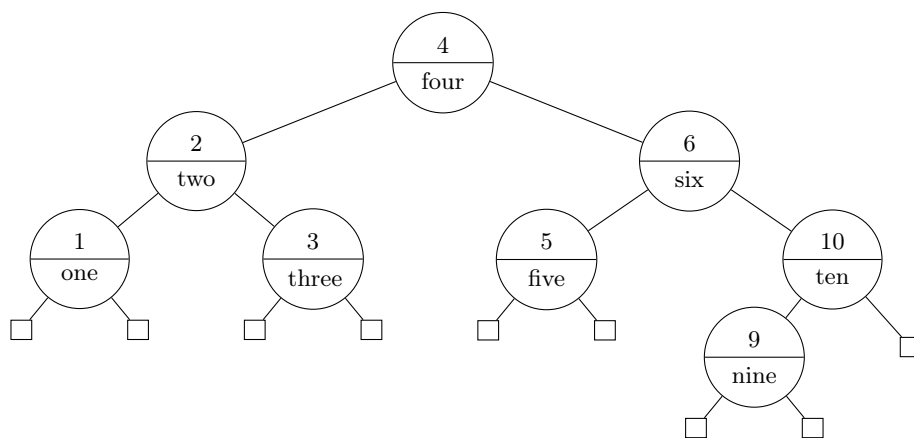
2.1 Definizione di Albero AVL

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

$$\forall x \in T \rightarrow |h(\text{left}(x)) - h(\text{right}(x))| \leq 1 \quad (*)$$

dove $h(k)$ indica il valore dell'altezza dell'albero radicato in k , e $\text{left}(B)$ (rispettivamente $\text{right}(B)$) 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, $\text{left}(\text{root})$ ha altezza 2, mentre $\text{right}(\text{root})$ ha altezza 3.

2.2 Implementazione della struttura dati

Come per la struttura dati degli Alberi binari di ricerca semplice, dobbiamo definire una classe `AVLNode` (sottoclasse di `Node`) per le istanze dei Nodi che compongono l'albero AVL:

```
1 class AVLNode(Node):
2     def __init__(self, value, str_name):
3         super().__init__(value, str_name)
4         self.height = 1
```

sources/Node.py

Una volta definita la classe `AVLNode`, possiamo procedere con l'implementazione della procedura per l'inserimento:

```
1 def avl_insert(root, value, str_name):
2     if root is None:
3         return AVLNode(value, str_name)
```

```

4     if value < root.key:
5         root.left = avl_insert(root.left, value, str_name)
6     else:
7         root.right = avl_insert(root.right, value, str_name)
8     root.height=1+max(getHeight(root.left),getHeight(root.right))
9     balance = getBalance(root)
10    # LL
11    if balance > 1 and value < root.left.key:
12        return rightRotate(root)
13    # RR
14    if balance < -1 and value > root.right.key:
15        return leftRotate(root)
16    # LR
17    if balance > 1 and value > root.left.key:
18        root.left = leftRotate(root.left)
19        return rightRotate(root)
20    # RL
21    if balance < -1 and value < root.right.key:
22        root.right = rightRotate(root.right)
23        return leftRotate(root)
24    return root

```

sources/avl.py

e con quella per la ricerca di un Nodo all'interno dell'Albero di tipo *AVL*:

```

1  def avl_find(root, value):
2      if root is None:
3          return
4      if root.key == value:
5          return root.name
6      if root.key < value:
7          return avl_find(root.right, value)
8      return avl_find(root.left, value)

```

sources/avl.py

3 Alberi binari di ricerca di tipo Red-Black

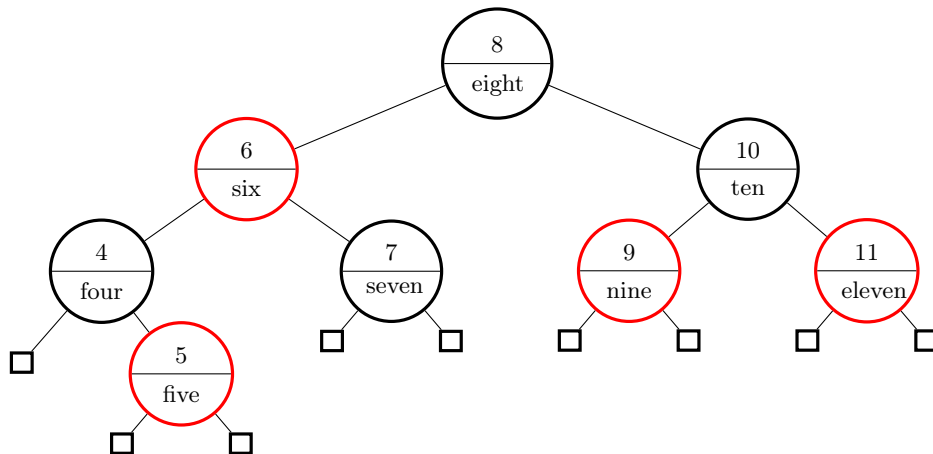
3.1 Definizione di *RB Tree*

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:

$$\forall x \in T \rightarrow h_b(\text{left}(x)) = h_b(\text{right}(x)) \quad (\bullet)$$

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 Un *BST* di tipo *Red-Black*, in cui ogni nodo contiene una chiave numerica dell'insieme $\{4, 5, 6, 7, 8, 9, 10, 11\}$ e un campo alfanumerico di tipo stringa, è il seguente:



3.2 Implementazione della struttura dati

Definiamo innanzitutto una classe `RBTNode` (sottoclasse di `Node`), in maniera analoga a quanto fatto per gli alberi di tipo *AVL*:

```
1 class RBTNode(Node):
2     def __init__(self, value, str_name):
3         super().__init__(value, str_name)
4         self.parent = None
5         self.color = "red"
```

sources/Node.py

Siamo quindi pronti per implementare in maniera ricorsiva la funzione di inserimento:

```
1 class RedBlackTree():
2     def __init__(self):
3         self.TNIL = RBTNode(None, None)
4         self.TNIL.color = "black"
5         self.TNIL.left = None
```

```

6         self.TNIL.right = None
7         self.root = self.TNIL
8
9
10    def left_rotate(self, x):
11        y = x.right
12        x.right = y.left
13        if y.left != self.TNIL:
14            y.left.parent = x
15        y.parent = x.parent
16        if x.parent == self.TNIL:
17            self.root = y
18        elif x == x.parent.left:
19            x.parent.left = y
20        else:
21            x.parent.right = y
22        y.left = x
23        x.parent = y
24
25
26    def right_rotate(self, x):
27        y = x.left
28        x.left = y.right
29        if y.right != self.TNIL:
30            y.right.parent = x
31        y.parent = x.parent
32        if x.parent == self.TNIL:
33            self.root = y
34        elif x == x.parent.right:
35            x.parent.right = y
36        else:
37            x.parent.left = y
38        y.right = x
39        x.parent = y
40
41
42    def rbt_insert(self, value, str_name):
43        z = RBTNode(value, str_name)
44        z.left = self.TNIL
45        z.right = self.TNIL
46        y = self.TNIL
47        x = self.root
48        while x != self.TNIL:
49            y = x
50            if z.key < x.key:
51                x = x.left
52            else:
53                x = x.right
54        z.parent = y

```



```
55         if y == self.TNIL:
56             self.root = z
57         elif z.key < y.key:
58             y.left = z
59         else:
60             y.right = z
61         self.insert_fix_up(z)
```

sources/rbt.py

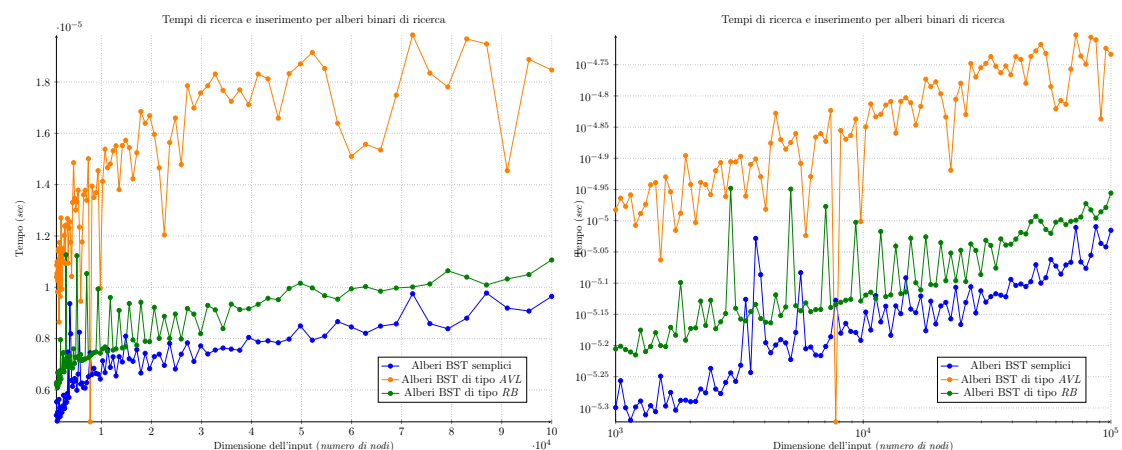
...

4 Calcolo della complessità

Implementate le tre strutture dati precedentemente descritte utilizzando il linguaggio Python, si è poi proceduto a calcolare i tempi medi per la ricerca e l'inserimento di n chiavi generate in modo pseudo-casuale.

4.1 Caso random

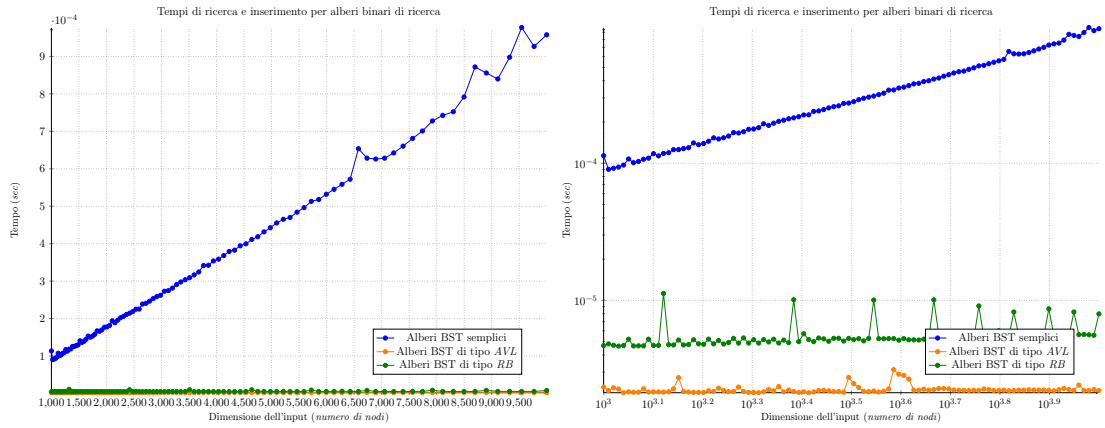
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Mauris dignissim vitae diam et placerat. Nulla egestas dui id ultrices hendrerit. Morbi in mi sed velit sollicitudin dictum ac vitae justo. Nunc auctor aliquam lectus nec aliquet. Aenean gravida elit sit amet pellentesque malesuada. Sed vel dolor porta, vestibulum nulla non, rutrum dolor. Morbi pretium mollis urna, at tempus erat aliquam ut. Ut id pellentesque elit. Etiam at dolor eget dui vehicula tempor. Nam mattis nec massa ut volutpat. Morbi id arcu sit amet augue faucibus eleifend in a metus.



Nullam eu nisl eget neque lacinia pulvinar. Maecenas sit amet aliquam metus. Etiam volutpat quam a mi varius, malesuada aliquet augue porttitor. Aenean ut imperdiet libero. In hac habitasse platea dictumst. Nulla ultrices consequat neque, vel dictum sem fringilla at. Phasellus in sapien sit amet lectus rutrum vestibulum id vitae neque. Suspendisse vitae felis vitae velit hendrerit mollis.

4.2 Caso sorted

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Mauris dignissim vitae diam et placerat. Nulla egestas dui id ultrices hendrerit. Morbi in mi sed velit sollicitudin dictum ac vitae justo. Nunc auctor aliquam lectus nec aliquet. Aenean gravida elit sit amet pellentesque malesuada. Sed vel dolor porta, vestibulum nulla non, rutrum dolor. Morbi pretium mollis urna, at tempus erat aliquam ut. Ut id pellentesque elit. Etiam at dolor eget dui vehicula tempor. Nam mattis nec massa ut volutpat. Morbi id arcu sit amet augue faucibus eleifend in a metus.



Nullam eu nisl eget neque lacinia pulvinar. Maecenas sit amet aliquam metus. Etiam volutpat quam a mi varius, malesuada aliquet augue porttitor. Aenean ut imperdiet libero. In hac habitasse platea dictumst. Nulla ultrices consequat neque, vel dictum sem fringilla at. Phasellus in sapien sit amet lectus rutrum vestibulum id vitae neque. Suspendisse vitae felis vitae velit hendrerit mollis.