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

21 maggio 2021

Indice

1 Introduzione **2**

2 Alberi binari di ricerca semplici **2**
2.1 Definizione di *BST* . . . 2
2.2 Implementazione della struttura dati . . . 3
2.2.1 Osservazioni sull'implementazione della struttura dati . . . 4

3 Alberi binari di ricerca di tipo AVL **5**
3.1 Definizione di Albero *AVL* . . . 5
3.2 Implementazione della struttura dati . . . 5

4 Alberi binari di ricerca di tipo Red-Black **7**
4.1 Definizione di *RB Tree* . . . 7
4.2 Implementazione della struttura dati . . . 7

5 Calcolo della complessità **9**
5.1 Caso random . . . 10
5.2 Caso sorted . . . 10
5.3 Conclusioni . . . 11

1 Introduzione

In questo elaborato si analizzano le implementazioni per tre strutture dati ad Alberi di ricerca, ovvero:

- gli Alberi binari di ricerca semplici,
- gli Alberi binari di ricerca di tipo AVL,
- gli Alberi binari di ricerca di tipo Red-Black.

Una volta implementate le strutture dati in Python3, si è proceduto all'analisi dei tempi medi di esecuzione per la ricerca e l'inserimento di n nodi per tipo di albero (con n ragionevolmente alto).

2 Alberi binari di ricerca semplici

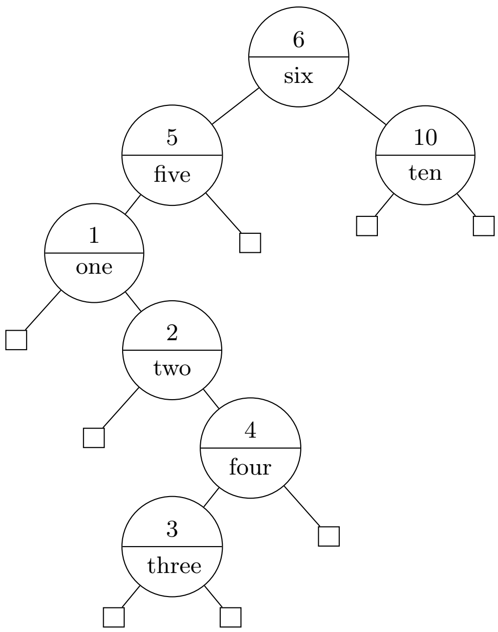
2.1 Definizione di *BST*

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

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

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.

2.2 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:

```
class Node(object):
    def __init__(self, value, str_name):
        self.key = value
        self.name = str_name
        self.left = None
        self.right = None
```

`sources/Node.py`

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

```
def bst_insert(root, value, str_name):
    if root is None:
        return Node(value, str_name)
    if value < root.key:
        root.left = bst_insert(root.left, value, str_name)
    else:
```

```
        root.right = bst_insert(root.right, value, str_name)
    return root
```

`sources/bst.py`

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

```
def bst_find(root, value):
    if root is None:
        return
    if root.key == value:
        return root.name
    if root.key < value:
        return bst_find(root.right, value)
    return bst_find(root.left, value)
```

`sources/bst.py`

2.2.1 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:

```
def bst_insert_iterative(root, value, str_name):
    newnode = Node(value, str_name)
    x = root
    y = None
    while (x != None):
        y = x
        if (value < x.key):
            x = x.left
        else:
            x = x.right
    if (y == None):
        y = newnode
    elif (value < y.key):
        y.left = newnode
    else:
        y.right = newnode
    return y
```

`sources/bst.py`

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

```
def bst_find_iterative(root, value):
    x = root
```

```python
    while x is not None:
        if x.key == value:
            return x.name
        if x.key < value:
            x = x.right
        else:
            x = x.left
    return
```

sources/bst.py

3 Alberi binari di ricerca di tipo AVL

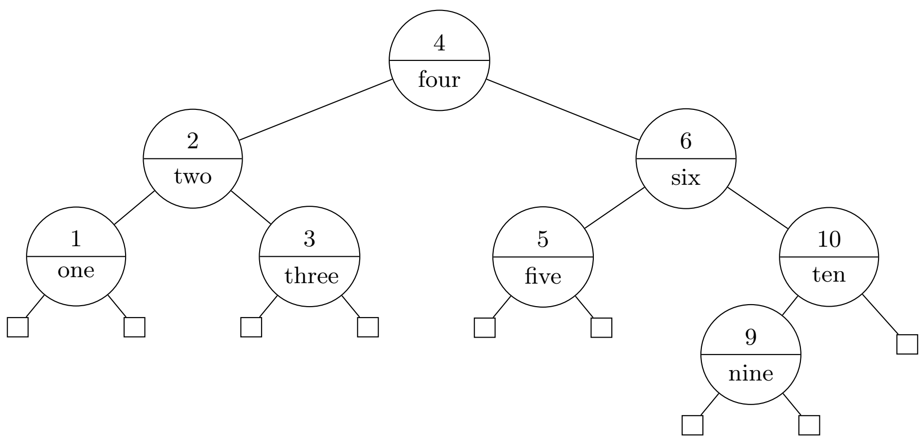
3.1 Definizione di Albero *AVL*

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

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

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, $left($`root`$)$ ha altezza 2, mentre $right($`root`$)$ ha altezza 3.

3.2 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:

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

sources/Node.py

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

```python
def avl_insert(root, value, str_name):
    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)
        return leftRotate(root)
    return root
```

sources/avl.py

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

```python
def avl_find(root, value):
    if root is None:
        return
    if root.key == value:
        return root.name
    if root.key < value:
        return avl_find(root.right, value)
    return avl_find(root.left, value)
```

sources/avl.py

4 Alberi binari di ricerca di tipo Red-Black

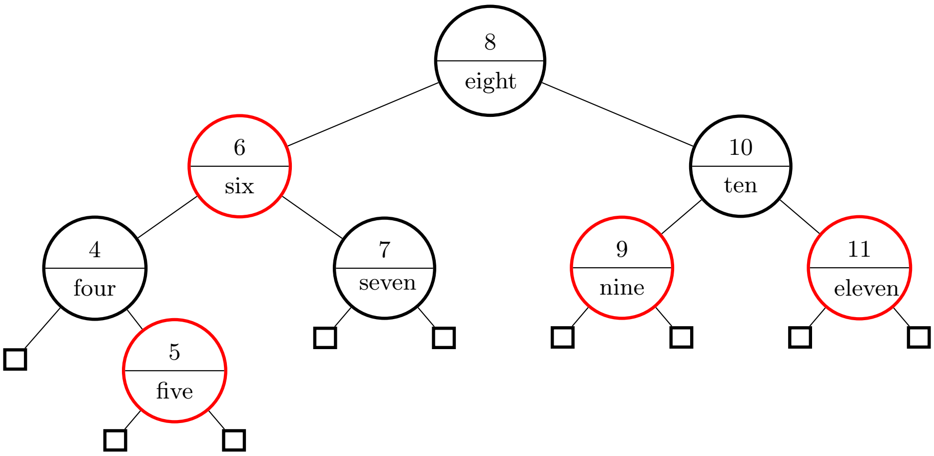
4.1 Definizione di *RB Tree*

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:

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

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 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:



4.2 Implementazione della struttura dati

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

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

`sources/Node.py`

Dobbiamo anche definire una classe `RedBlackTree`, per gestire le foglie `NIL`:

```
class RedBlackTree():
    def __init__(self):
        self.TNIL = RBTNode(None, None)
        self.TNIL.color = "black"
        self.TNIL.left = None
```

```
        self.TNIL.right = None
        self.root = self.TNIL
```

sources/rbt.py

Siamo quindi pronti per implementare la funzione di inserimento

```
    def rbt_insert(self, value, str_name):
        z = RBTNode(value, str_name)
        z.left = self.TNIL
        z.right = self.TNIL
        y = self.TNIL
        x = self.root
        while x != self.TNIL:
            y = x
            if z.key < x.key:
                x = x.left
            else:
                x = x.right
        z.parent = y
        if y == self.TNIL:
            self.root = z
        elif z.key < y.key:
            y.left = z
        else:
            y.right = z
        self.insert_fix_up(z)


    def insert_fix_up(self, z):
        while z.parent.color == "red":
            if z.parent == z.parent.parent.right:
                y = z.parent.parent.left
                if y.color == "red":
                    y.color = "black"
                    z.parent.color = "black"
                    z.parent.parent.color = "red"
                    z = z.parent.parent
                else:
                    if z == z.parent.left:
                        z = z.parent
                        self.right_rotate(z)
                    z.parent.color = "black"
                    z.parent.parent.color = "red"
                    self.left_rotate(z.parent.parent)
            else:
                y = z.parent.parent.right
                if y.color == "red":
                    y.color = "black"
                    z.parent.color = "black"
```

```
                    z.parent.parent.color = "red"
                    z = z.parent.parent
                else:
                    if z == z.parent.right:
                        z = z.parent
                        self.left_rotate(z)
                    z.parent.color = "black"
                    z.parent.parent.color = "red"
                    self.right_rotate(z.parent.parent)
            if z == self.root:
                break
        self.root.color = "black"
```

sources/rbt.py

e la funzione di ricerca

```
def rbt_find(root, value):
    if root.key is None:
        return
    if root.name is None:
        return
    if root is None:
        return
    if root.key == value:
        return root.name
    if root.key < value:
        return rbt_find(root.right, value)
    return rbt_find(root.left, value)
```

sources/rbt.py

5 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.

Nello specifico, per ogni struttura dati si sono generati 100 input, le cui dimensioni seguono una distribuzione geometrica tra la dimensione 1 000 e la dimensione 100 000, e singoli valori generati in maniera casuale. Si è poi proceduto calcolando la risoluzione del sistema (pari, nel nostro calcolatore, a circa 0.9547 millesimi di secondo). Per ultimo, si è memorizzato in un array i tempi (ammortizzati nella risoluzione del sistema) di esecuzione della ricerca e successivo inserimento per ogni input, come mostrato di seguito (solo per gli Alberi Binari di Ricerca di tipo semplice):

```
amortizedTimes_BST = []
for n in dim:
    i = 0
```

```
    t_passed = 0
    while ( t_passed <= t_min ):
        start = t.time()
        runBSTrandom(n)
        end = t.time()
        i += 1
        t_passed += end-start
    amortizedTimes_BST.append((t_passed/i)/n)
```

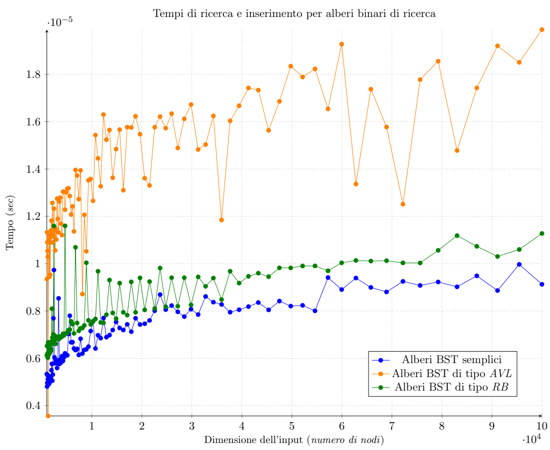
sources/complexity.py

```
def runBSTrandom(n): # int n -> size of the tree
    node = Node(random.randint(0, MAX_KEY), "")
    for i in range(0, n):
        key = random.randint(0, MAX_KEY)
        if bst_find(node, key) == None:
            bst_insert(node, key, "")
```

sources/complexity.py

5.1 Caso random

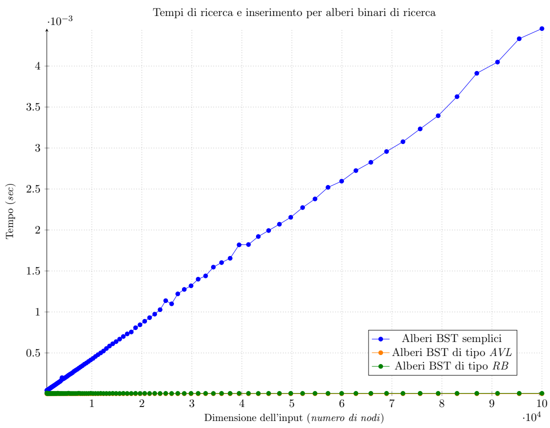
I grafici dei risultati ottenuti sono illustrati di seguito, sia in scala lineare che in scala logaritmica.

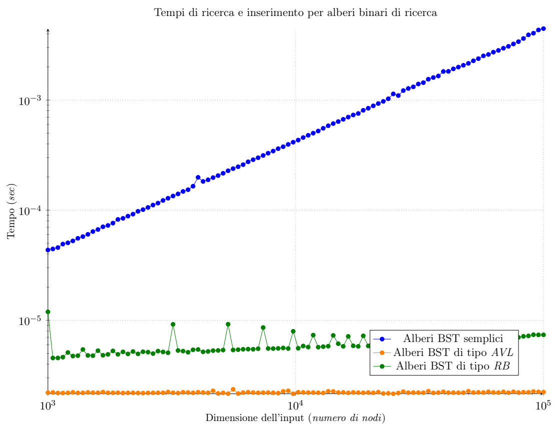


La struttura dati che globalmente ottiene prestazioni peggiori è quella degli Alberi Binari di Ricerca di tipo AVL, mentre quelli semplici e di tipo RB ottengono risultati pressoché simili. Si nota, comunque, che anche nel caso di maggiore distanza in termini di tempo tra gli alberi semplici e RB, e gli alberi AVL la differenza è dell'ordine dei microsecondi.

5.2 Caso sorted

Si è proceduto inoltre ad analizzare i tempi medi di esecuzione nelle tre strutture dati anche nel caso di input che rappresenti il caso pessimo possibile: in questo caso, lo scenario peggiore per le funzioni di inserimento è quello di input ordinati in ordine crescente (ma sarebbe stato equivalente considerare l'ordine decrescente). Questo perché, per inserire un nuovo nodo nell'albero, dobbiamo necessariamente scorrere tutto il percorso verso la sua posizione finale. Nel caso di input ordinati, appunto, questo percorso sarà sempre massimizzato.





Come ci si aspettava, la struttura dati che ha registrato tempi di esecuzione decisamente maggiori è quella degli Alberi Binari di ricerca semplici, dato che non introducono alcuna politica per il bilanciamento dell'albero in fase di popolazione dello stesso.

Gli alberi di tipo *Red-Black* ottengono pressapoco lo stesso risultato del caso di input randomici, mentre gli alberi di tipo *AVL* risultano la struttura dati che questo caso ottiene risultati migliori, ma comunque poco distanti dai *Red-Black*.

5.3 Conclusioni

In conclusione, si ritiene che la struttura dati che globalmente è maggiormente indicata per la memorizzazione di alberi binari di ricerca, senza porre limitazioni alla struttura degli input, siano proprio gli alberi di tipo *Red-Black*.