



UNIVERSITÀ
DEGLI STUDI
FIRENZE

Alberi Binari di Ricerca con Chiavi Duplicate

Niccolò Caselli

20/01/2024

Relazione di Laboratorio di Algoritmi e Strutture Dati

Indice

1	Introduzione	2
2	Cenni sugli Alberi Binari di Ricerca	2
2.1	Cos'è un albero binario di Ricerca?	2
2.2	Implementazione in Python	3
3	Il Problema delle Chiavi Duplicate	4
3.1	Implementazione "normale"	4
3.2	Metodo del Flag Booleano	5
3.3	Metodo della Lista Concatenata	5
4	Test delle prestazioni	5
4.1	Analisi Teorica	5
4.2	Risultati Sperimentali	5
	Nota sulle Caratteristiche del Computer	6
	Bibliografia	6

1 Introduzione

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

2 Cenni sugli Alberi Binari di Ricerca

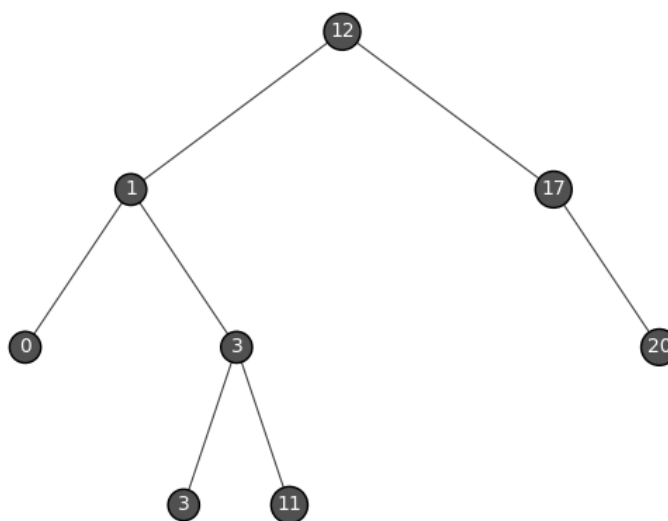


Figura 1: Esempio di albero binario di ricerca

2.1 Cos'è un albero binario di Ricerca?

Prima di esaminare il problema delle chiavi duplicate cerchiamo di capire cos'è un albero binario di ricerca (abbreviato spesso in B.S.T dall'inglese). Un BST è un tipo di struttura dati basata su gli alberi, definiti nel libro *Introduzione agli algoritmi e strutture dati* come "grafi non orientati, connessi e aciclici". Tale struttura ha la caratteristica di permettere le operazioni di base in $O(h)$, con h l'altezza dell'albero, che gli rende efficienti per la realizzazione di dizionari, ricerca di valori, e attraversamenti.

Un albero binario di ricerca è costituito da una serie di nodi che nel calcolatore possono essere rappresentati con oggetti con i seguenti attributi:

- una chiave ($x.key$)
- un puntatore al figlio sinistro ($x.left$)
- un puntatore al figlio destro ($x.right$)
- un puntatore al padre ($x.p$)

Enunciamo quindi la proprietà fondamentale di un albero binario di ricerca:

Proprietà (2.1)

- Se y è nel sottoalbero sinistro di x , allora $y.key < x.key$
- Se y è nel sottoalbero destro di x , allora $y.key > x.key$

Tale caratteristica dei BST è facilmente individuabile nell'albero di esempio di Figura 1.

2.2 Implementazione in Python

Come già accennato possiamo rappresentare un albero binario con l'ausilio delle classi. Creiamo quindi una classe per rappresentare l'intero albero e una classe per il singolo nodo dell'albero. Tra queste due classi vi è quindi un relazione *one-to-many*.

Codice 1: Implementazione di un BST

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

class BST:
    def __init__(self):
        self.root = None

    # Imposta la radice dell'albero
    def setRoot(self, key):
        self.root = Node(key)

    # Inserisce un nodo nell'albero
    def insert(self, key):
        if (self.root is None):
            self.setRoot(key)
        else:
            self._insertNode(self.root, key)

    # Funzione di supporto per l'inserimento di un nodo
    def _insertNode(self, currentNode, key):
        if (key < currentNode.key):
            if (currentNode.left):
                self._insertNode(currentNode.left, key)
            else:
                currentNode.left = Node(key)
        elif (key > currentNode.key):
            if (currentNode.right):
                self._insertNode(currentNode.right, key)
            else:
                currentNode.right = Node(key)
```

Per ovvi motivi di impaginazione è stato riportato solo parte del codice; il lettore può quindi estenderlo a suo piacere implementando i metodi per gli attraversamenti dell'albero e la ricerca. Come già sottolineato (Vedi 2.1) il vantaggio di questa struttura dati è il costo asintotico delle sue operazioni di base. Più nello specifico, è evidente dalla funzione precedentemente mostrata, l'inserimento in un albero binario è un'operazione che richiede $O(h)$ (dal momento che prima di inserire un nuovo nodo bisogna raggiungere una foglia percorrendo al più tutta l'altezza dell'albero - h). Lo stesso discorso si applica per la ricerca e cancellazione (risulta invece diverso il caso degli attraversamenti - in ordine, posticipato e anticipato - che richiedono $\Theta(n)$). Questo aspetto è fondamentale poiché spiega l'esigenza, su cui torneremo, di avere alberi il più possibili bilanciati. È infine degna di nota la funzione ricorsiva usata nella relazione per la rappresentazione grafica dei BST.

Codice 2: Rappresentazione Grafica di un BST

```

def plot_tree(tree):
    fig, ax = plt.subplots()
    _plot_tree(ax, tree.root, x=0, y=0, level=1)
    ax.axis('off')
    plt.show()

def _plot_tree(ax, node, x, y, level):
    if node is not None:
        ax.annotate(node.key, (x, y), xytext=(x, y),
                    color="white",
                    ha='center',
                    va='center',
                    bbox=dict(boxstyle='circle', fc='#505050'))

    xfactor = 1/2
    y_new = level * -2

    if node.left is not None:
        x_new = x - xfactor * level
        ax.plot([x, x_new], [y, y_new], linewidth=.8, color="#404040")
        _plot_tree(ax, node.left, x_new, y_new, level + 1)

    if node.right is not None:
        x_new = x + xfactor * level
        ax.plot([x, x_new], [y, y_new], linewidth=.8, color="#404040")
        _plot_tree(ax, node.right, x_new, y_new, level + 1)

```

3 Il Problema delle Chiavi Duplicate

La definizione data fin'ora di Albero Binario di Ricerca si basa sulla fondamentale supposizione di unicità delle chiavi (come implicito nella proprietà 2.1). Sappiamo però che questa ipotesi è difficilmente riscontrabile nelle applicazioni reali: entriamo, dunque, nel merito del problema delle chiavi duplicate.

Possiamo trovare vari possibili approcci per gestire il problema, in questa relazione ne verranno discussi tre:

- Implementazione *normale* (3.1).
- Implementazione con Flag Booleano (3.2).
- Implementazione con Lista Concatenata (3.3).

3.1 Implementazione "normale"

Un primo possibile approccio, e senza dubbi il più immediato, è quello di aggiustare la definizione, volutamente ambigua nel caso di chiavi duplicate, fornita precedentemente (vedi proprietà 2.1).

Possiamo quindi riformulare la relazione tra i nodi di un BST affinché supporti l'inserimento quando la chiave da inserire è già presente nell'albero:

Proprietà (3.1)

- Se y è nel sottoalbero sinistro di x , allora $y.key \leq x.key$
- Se y è nel sottoalbero destro di x , allora $y.key > x.key$

Questa modifica comporta quindi un cambiamento dell'implementazione dell'inserimento nella classe BST: al momento dell'inserimento i nodi con stessa chiave vengono inseriti come figli sinistri del primo nodo con tale chiave (e così via ricorsivamente). Aggiustiamo quindi il codice.

Codice 3: Funzione ausiliaria di inserimento aggiornata

```

def _insertNode(self, currentNode, key):
    if (key <= currentNode.key):
        if (currentNode.left):
            self._insertNode(currentNode.left, key)
        else:
            currentNode.left = Node(key)
    elif (key > currentNode.key):
        if (currentNode.right):
            self._insertNode(currentNode.right, key)
        else:
            currentNode.right = Node(key)

```

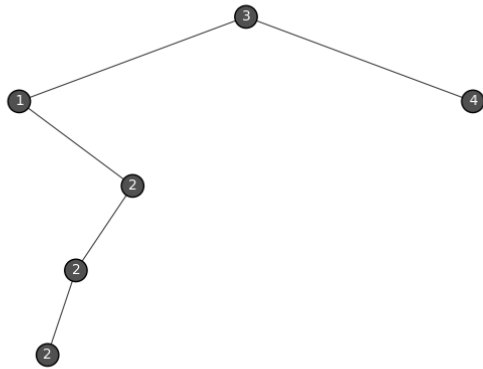


Figura 2: BST con chiavi duplicate

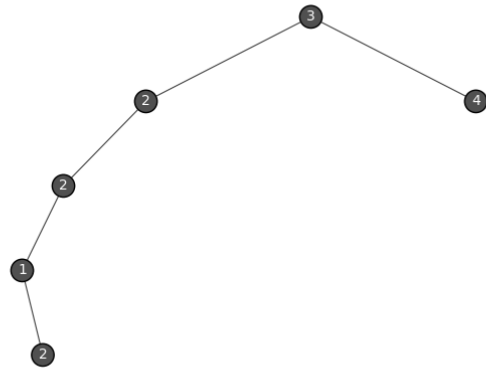


Figura 3: Altro esempio di BST con chiavi duplicate

Nella Figura 2 possiamo vedere l'albero derivante dall'inserimento ordinato dei seguenti elementi $\{3, 1, 2, 2, 2, 4\}$. Notiamo quindi che vi sono tre nodi con la stessa chiave 2, ognuno dei quali è nel sotto-albero sinistro del nodo padre più vicino con la medesima chiave.

Non bisogna però cadere nella convinzione che nodi con stessa chiave si trovino sempre tutti adiacenti l'uno con l'altro. A questo scopo si consideri la Figura 3: cambiando semplicemente l'ordine di inserimento dei valori della figura precedente (ora $\{3, 2, 2, 1, 2, 4\}$) il risultato è molto diverso; tra il penultimo e l'ultimo nodo con chiave 2 vi è in mezzo un nodo con chiave 1! Questo può portare a complicazioni nella ricerca, che verrà affrontata in seguito.

3.2 Metodo del Flag Booleano

3.3 Metodo della Lista Concatenata

4 Test delle prestazioni

4.1 Analisi Teorica

4.2 Risultati Sperimentali

Nota sui test

Si tenga presente che tutti i test presenti in tale relazione sono stati eseguiti su un calcolatore con le seguenti specifiche:

Hardware:

- **Modello:** MacBook Pro 2022
- **CPU:** Apple Silicon M2 (8 core)
- **RAM:** 8 GB LPDDR5 (Micron)

Software:

- **Versione macOS:** 14.0
- **Versione Python:** 3.11.4
- **Versione Pip3:** 23.3.2

Bibliografia

Cormen, Thomas H., Charles E. Leiserson e Ronald L. Rivest. *Introduzione agli algoritmi e strutture dati*. McGraw-Hill Education, 2023.