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

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer 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

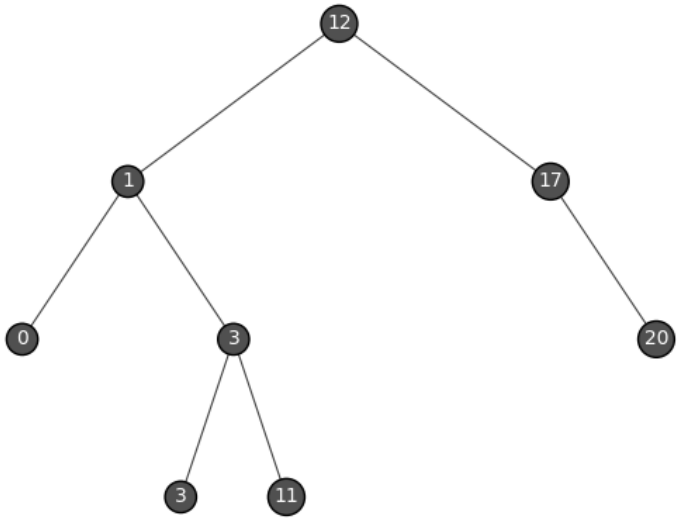

Figura 1: Esempio di albero binario di ricerca

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 Grfaica 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)
```

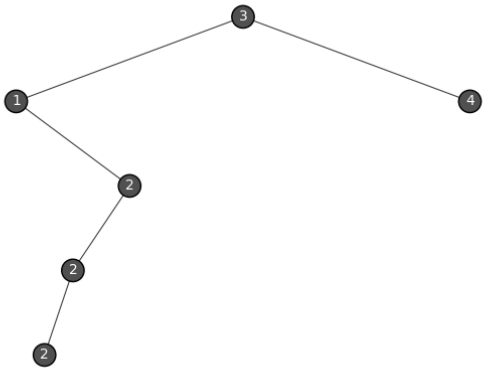

Figura 2: BST con chiavi duplicate

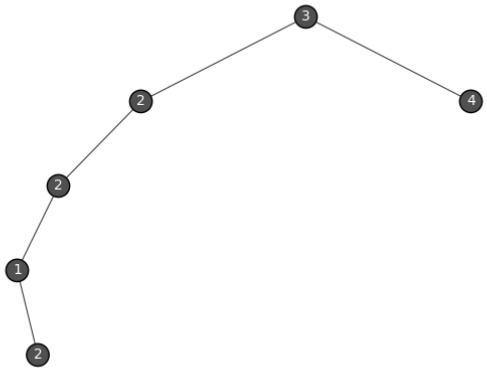

Figura 3: Altro esempio di BST con chiavi duplicate

Nella Figura 2 possiamo vedere l'albero derivante dall'inserimento ordinato dei seguenti elementi $\{\mathbf{3}, \mathbf{1}, \mathbf{2}, \mathbf{2}, \mathbf{2}, \mathbf{4}\}$. 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 $\{\mathbf{3}, \mathbf{2}, \mathbf{2}, \mathbf{1}, \mathbf{2}, \mathbf{4}\}$) 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.

Il principale svantaggio di questa implementazione è che l'inserimento di tante chiavi uguali può tendere a sbilanciare l'albero il quale finisce per degenerare in una semplice lista. Infatti, è evidente che per creare l'albero di N elementi più sbilanciato possibile sia sufficiente inserire N volte la stessa chiave. Come in esempio (Figura 4).

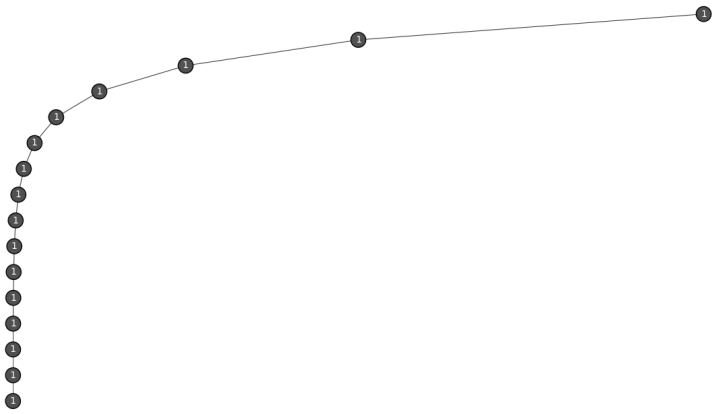

Figura 4: Esempio di albero binario di ricerca sbilanciato

3.2 Metodo del Falg Booleano

Un altro possibile approccio al caso delle chiavi duplicate è, invece di inserire sempre i nodi con chiave duplicate a sinistra del nodo padre, inserire i nodi con chiave duplicate a sinistra o destra del padre a seconda di un flag booleano. Più nello specifico: estendiamo la struttura dati di ogni nodo dell'albero così che il generico nodo x abbia un nuovo attributo *x.flag* e quindi, durante l'inserimento, porre x a *x.left* o *x.right* a seconda del valore del flag, il quale viene alternato a ogni visita di x con chiave uguale.

Il punto di forza di questa implementazione è il fatto che tende a preservare il bilanciamento dell'albero anche in presenza di molte chiavi duplicate, a differenza del primo approccio (Vedi 3.1). Ciò risulta evidente se si prova a disegnare l'albero risultante dall'inserimento di N chiavi uguali (Vedi Figura 5).

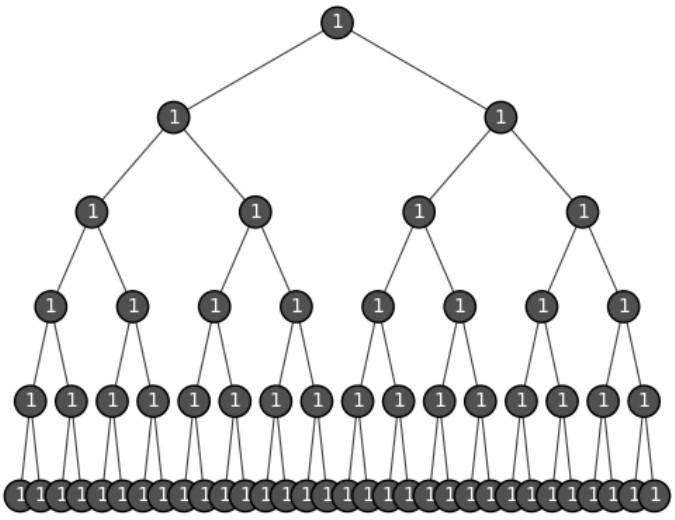

Figura 5: BST con Flag Booleano di 64 chiavi uguali

Per quanto riguarda il codice, le modifiche da apportare sono minime. Aggiungiamo il nuovo attributo nella classe Nodo:

Codice 4: Classe nodo di un BST con Flag Booleano

```
class Node:
    def __init__(self, key):
```

```
        self.key = key
        self.left = None
        self.right = None
        self.flag = False # <--
```

E aggiorniamo la funzione di supporto per l'inserimento:

```
    def _insertNode(self, currentNode, key):
        if(currentNode.key == key):
            if(currentNode.flag): self._insert_right(currentNode, key)
            else: self._insert_left(currentNode, key)
            currentNode.flag = not currentNode.flag # Alterna il flag

        if (key < currentNode.key): self._insert_left (currentNode, key)
        elif (key > currentNode.key): self._insert_right(currentNode, key)

    # Funzione di supporto per l'inserimento di un nodo a sinistra
    def _insert_left(self, currentNode, key):
        if (currentNode.left): self._insertNode(currentNode.left, key)
        else: currentNode.left = Node(key)

    # Funzione di supporto per l'inserimento di un nodo a destra
    def _insert_right(self, currentNode, key):
        if (currentNode.right): self._insertNode(currentNode.right, key)
        else: currentNode.right = Node(key)
```

3.3 Metodo della Lista Concatenata

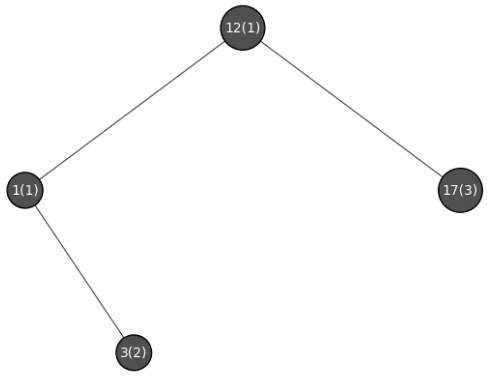

Figura 6: BST con gestione chiavi duplicate tramite lista concatenata

Figura 7: BST con lista concatenata di 100 chiavi uguali

L'ultimo approccio possibile trattato in questa relazione è quello della lista concatenata: inseriamo tutti i nodi con chiave uguale in una lista concatenata. Una possibile implementazione nel codice è di estendere la classe Nodo con un puntatore *next* al prossimo elemento della lista concatenata. Nel mio caso ho aggiunto anche un capo *duplicates* che, nel primo nodo, tiene il conteggio dei nodi con chiave duplicata per esigenze rappresentative.

Codice 5: Classe Nodo di un BST con lista concatenata

```
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.next = None # <--
        self.duplicates = 0 # <--
```

A questo punto, in fase di inserimento, se si trova un nodo con stessa chiave già esistente basta inserire il nuovo nodo.

Codice 6: Gestione chiavi duplicate con lista concatenata

```
if (key == currentNode.key):
        newNode = Node(key)
        newNode.next = currentNode.next
        currentNode.next = newNode
        currentNode.duplicates += 1
```

Apportando una semplice modifica alla funzione *plot_tree* (Vedi codice 2) possiamo visualizzare questo approccio in Figura 6 e Figura 7. Per ogni nodo è segnato tra parentesi il numero di occorrenze di quella chiave.

4 Test delle prestazioni

Entriamo ora nel dettaglio sulle prestazioni di queste tre possibili implementazioni. Esaminiamo quindi il problema dell'inserimento di n chiavi identiche all'interno di un BST inizialmente vuoto e l'operazione di ricerca.

4.1 Analisi Teorica

Come già discusso nel paragrafo 3.1 l'implementazione senza particolari accorgimenti ha il problema della degenerazione dell'albero in una lista del caso di un elevato numero di inserimento con stessa chiave. La teoria ci dice infatti che l'inserimento in un albero di ricerca è un operazione che richiede $O(h)$; tuttavia, come immaginabile dalla Figura 4, nell'inserimento di n chiavi identiche si deve ogni volta arrivare in fondo all'albero di altezza $h = n$. Ne consegue che ci si aspetta un comportamento lineare $\Theta(n)$.
Il secondo approccio (Paragrafo 3.2) evita questa problematica e cerca di preservare il bilanciamento dell'albero. L'altezza diventa quindi funzione logaritmica e la complessità per l'inserimento $\Theta(log(n))$.
Infine per quanto riguarda l'inserimento con l'approccio tramite lista concatenata (Vedi 3.3) sappiamo che una volta trovato il primo nodo con chiave uguale, l'inserimento in testa alla lista concatenata è un'operazione $\Theta(1)$; dobbiamo però prima trovare tale nodo, il quale può anche non esistere. Abbiamo allora $O(h) + \Theta(1) = O(h)$. Tuttavia, nel caso del test di inserimento di n chiavi uguali l'albero degenera in un singolo nodo (Vedi Figura 7) e quindi la complessità risultante diventa $\Theta(1) + \Theta(1) = \Theta(1)$

4.2 Risultati Sperimentali

Nota sui test

Si tenga presente svolgere i test discussi è necessario alzare il limite massimo di ricorsione di default di Python. Questo è necessario per prevenire che l'interprete blocchi l'esecuzione di un elevato numero di chiamate ricorsive, però necessarie in questa tipologia di test. Per fare ciò basta impostare un nuovo limite nel seguente modo:

```
sys.setrecursionlimit(500000000)
```

Si tenga inoltre 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.