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 Premessa sulla ricerca 4
3.2 Implementazione "normale" 4
3.3 Metodo del Falg Booleano 6
3.4 Metodo della Lista Concatenata 7

4 Test delle prestazioni **8**
4.1 Analisi Teorica 8
4.2 Risultati Sperimentali 8

Nota sui test **12**

Bibliografia **12**

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

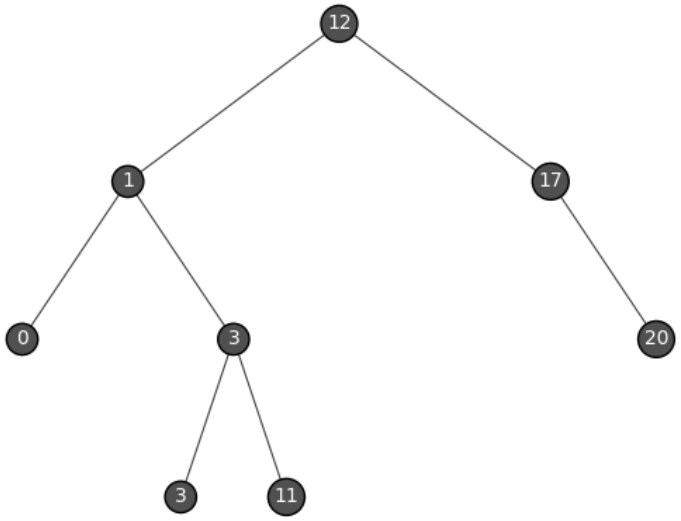

Figura 1: Esempio di albero binario di ricerca

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* (3.2).
- Implementazione con Flag Booleano (3.3).
- Implementazione con Lista Concatenata (3.4).

3.1 Premessa sulla ricerca

Nonostante i test eseguiti e la relazione si concentreranno sull'inserimento, è comunque necessario fare una premessa su cosa intendiamo come ricerca in un Albero Binario di Ricerca con chiavi duplicate. A seconda del tipo di utilizzo che vogliamo dare a un BST possiamo infatti realizzare diversamente la ricerca. In alcuni testi la ricerca di un nodo è descritta come un metodo che restituisce semplicemente se è presente o meno una specifica chiave in un albero o, in alternativa, che restituisce il primo nodo con tale chiave. In questi due specifici casi non bisogna mettere in pratica nessuna modifica nella implementazione precedente, tuttavia ciò è molto limitante: sebbene per motivi didattici quando si tratta i BST si tende a semplificare considerando ogni nodo solo dalla sua chiave, nella realtà, ogni nodo contiene anche dati satelliti. Ne consegue che sebbene due nodi possano avere una stessa chiave, il loro valore può essere diverso. È il caso ad esempio dei dizionari; se in Javascript e Python non sono supportate le chiavi duplicate nei dizionari (oggetti in Javascript), un esempio valido sono invece le *multimap* del c++ le quali, data una chiave, restituiscono tutti i valori associati a tale chiave. Questo è l'approccio seguito in questa relazione.

3.2 Implementazione "normale"

Un primo possibile approccio alla questione delle chiavi duplicate, e senza dubbi quello più immediato, è semplicemente aggiustare la definizione, volutamente ambigua, 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)
```

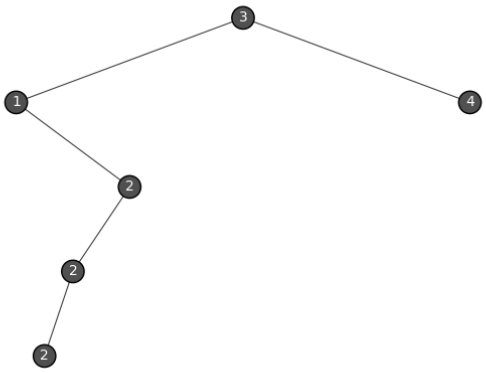

Figura 2: BST con chiavi duplicate

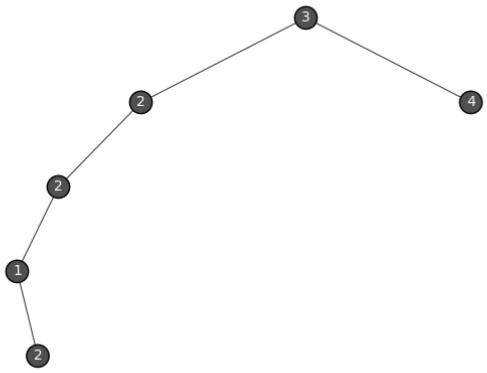

Figura 3: Altro esempio di BST con chiavi duplicate

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).

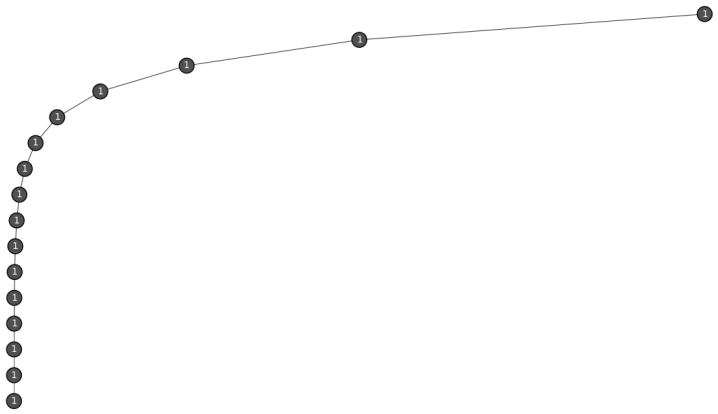

Figura 4: Esempio di albero binario di ricerca sbilanciato

3.3 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 3.2). Ciò risulta evidente se si prova a disegnare l'albero risultante dall'inserimento di N chiavi uguali (Vedi Figura 5).

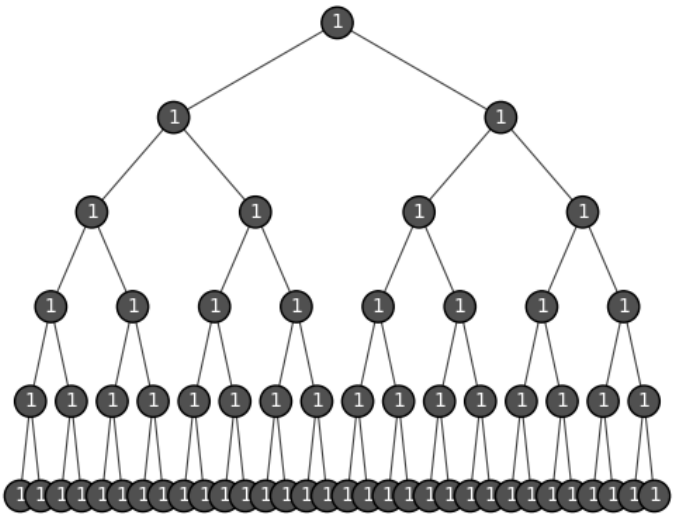

Figura 5: BST con Flag Booleano di 64 chiavi uguali

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)
```

3.4 Metodo della Lista Concatenata

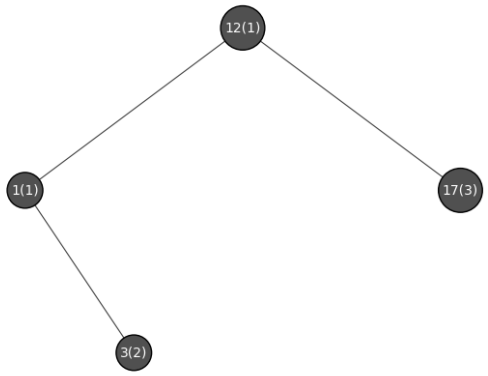

Figura 6: BST con gestione chiavi duplicate tramite lista concatenata

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 3.2 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 3.3) 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 3.4) 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

Come già detto il test eseguito consiste nell'inserire N (numero molto grande) volte la stessa medesima chiave in un albero binario di ricerca per comprendere in questo caso estremo le differenze tra le tre implementazioni.

Il primo approccio, come si ipotizzato nel paragrafo precedente, si è rilevato il più dispendioso di risorse computazionali e di tempo. Infatti, se ii grafici generarti per gli altri due casi arrivano fino all'inserimento in Alberi Binari di Ricerca di fino un milione di elementi, nel caso dell'implementazione classica del BST il test è stato eseguito fino a un albero di 200-mila elementi. Come apprezzabile dal grafico riportato, il costo di questa operazione segue un andamento lineare e già i vari test fino a 200.000 chiavi hanno impiegato sul mio calcolatori (Vedi la nota sui test) più di mezz'ora di esecuzione. I test su N nell'ordine dei milioni, che ho più volte tentato, era semplificante non praticabile, e a mio avviso nemmeno significativo. Per i test è stata inoltre apportata una modifica alla classe Albero per tenere traccia del numero di iterazioni (in realtà se si vuole essere precisi, ricorsioni) che sono state necessarie per l'inserimento di ogni valore in un albero di dimensione n. Ad ogni dimensione dell'albero, quindi, sono stati salvati i tempi di esecuzione e il numero di iterazioni impiegati in un dizionario, ovvero un hash table di Python, utilizzando il valore corrente di n come chiave. A conferma dell'andamento lineare dell'inserimento con il primo approccio è quindi anche il grafico del numero di interazioni e, sopratutto, la tabella (2) che ci mostra chiaramente l'identità tra il numero di elementi dell'albero e il numero di interazioni necessarie per l'inserimento (questo ovviamente perché necessario scorrere tutto l'albero degenerato in una lista concatenata).

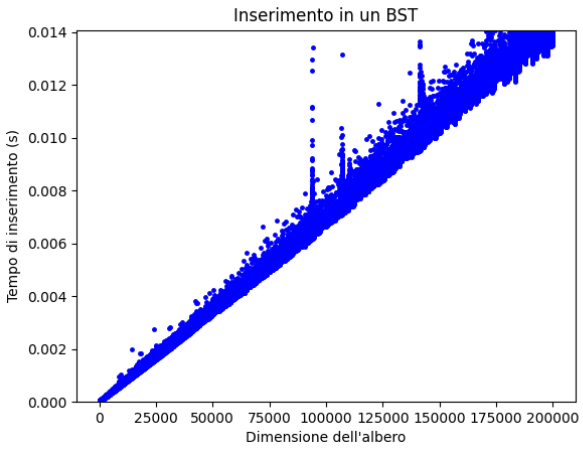

Figura 8: Prestazioni di inserimento di 200.000 chiavi identiche un BST senza accorgimenti particolari

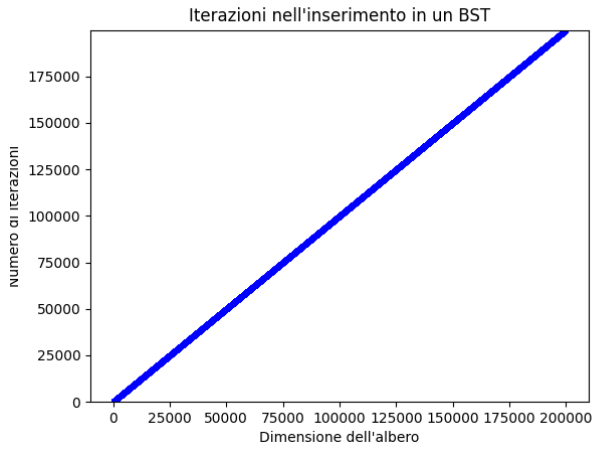

Figura 9: Variazione del numero di iterazioni per l'inserimento in un BST di 200.000 chiavi identiche

Numero di elementi	Tempo di esecuzione (s)
0	2.29e-06
15384	1.02e-03
30768	2.10e-03
46152	3.11e-03
61536	4.17e-03
76920	5.13e-03
92304	6.11e-03
107688	7.72e-03
123072	8.58e-03
138456	9.75e-03
153840	1.10e-02
169224	1.23e-02
184608	1.33e-02
199992	1.37e-02
199998	1.38e-02

Tabella 1: Risultati dell'inserimento in un BST normale

Numero di elementi	Iterazioni
0	0
15384	15384
30768	30768
46152	46152
61536	61536
76920	76920
92304	92304
107688	107688
123072	123072
138456	138456
153840	153840
169224	169224
184608	184608
199992	199992
199998	199998

Tabella 2: Numero di iterazioni nell'inserimento in un BST normale

Nota sulle tabelle. I test sono stati eseguiti su alberi di dimensioni fino a un milione di elementi e sono stati salatavi i risultati in file *.xlsx* da migliaia di righe. Ne consegue che, per ovvi motivi, questi file sono stati successivamente abbreviati selezionando solo le misurazioni più significative da un altro script Python.

L'approccio del Flag Booleano da un punto di vista teorico dovrebbe dimostrarsi caratterizzato da una complessità logaritmica, tuttavia in questo caso diventa già molto più difficile interpretare il grafico. Infatti, quando si parla di logaritmo spesso si tende a sottostimare quando questa funzione cresci lentamente: nel caso di 10.000 elementi sono necessarie solo 13 iterazioni per arrivare in fondo all'albero e inserire un nuovo valore; con un milione di elementi si sale ad appena a 19 iterazioni. Risulta evidente che per un calcolatore moderno la differenza, ad esempio, tra 13 o 19 iterazioni (e quindi di fatto confronti) è praticamente impercettibile e quindi il grafico risulta quasi costante e indistinguibile con quello del *BST LIST*. I test sono stati eseguiti su 1 milione di elementi e hanno impiegato un tempo quasi insignificante. Tuttavia, cercando di aumentare il numero di elementi ho riscontrato dei problemi con il terminale e l'editor di testo che sono andati in crash. Suppongo però, mettendo le mani aventi poiché non ne posso avere la certezza, che questo sia un problema specifico dell'IDE, o della gestione del sistema o di Python delle chiamate ricorsive.

Nonostante tutto ciò, dopo aver messo in atto accorgimenti alla visualizzazione del grafico e zoomando nella sezione di interesse, è possibile scorgere l'andamento logaritmico, anche se con molto rumore (dovuto probabilmente ad altri processi del calcolatore e in generale all'aleatorietà di tale tipo di test quando si va a considerare intervalli di tempo così ridotti). Questo risulta ancora pià evidente se si prende in considerazione il grafico delle iterazioni di inserimento.

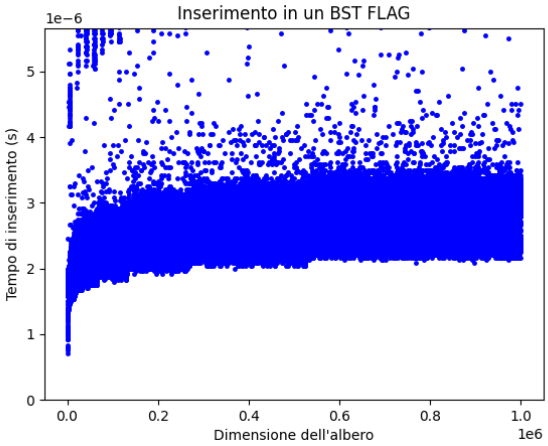

Figura 10: Prestazioni di inserimento di 1.000.000 chiavi identiche un BST con Flag booleano

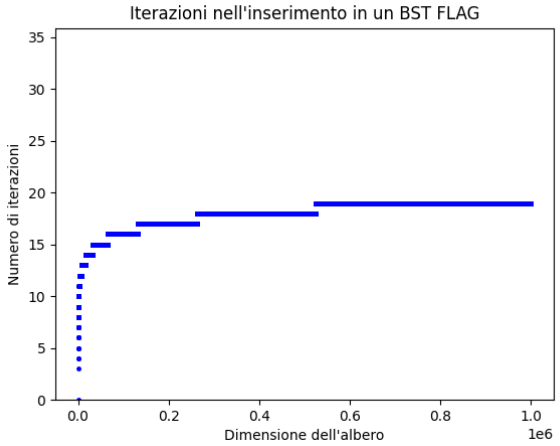

Figura 11: Iterazioni nell'inserimento di 1.000.000 chiavi identiche un BST con Flag booleano

L'approccio della Lista Concatenata deve avere come grafico un andamento costante, e così è. Nuovamente, risulta tuttavia di non immediata interpretazione: infatti, se rappresentiamo il grafico concentrandosi sulla zona d interesse, come nel caso di Figura 12, essendo che in questo caso l'operazione di inserimento è molto veloce e poco dispendiosa di risorse, si ha che la variabile di incertezza/casualità influenza maggiormente il risultato. Ecco perché il grafico in Figura 12 non risulta perfettamente come costante, ma i valori dei test si distribuiscono quasi uniformemente tra un range di valori. È inoltre fondamentale notare come la scala verticale dei grafici di Figura 12 e Figura 10 sia dversa; rispettivamente dell'ordine di 10e-6 per la prima e 10e-7 per la seconda, che quindi risulta più "zoomata". Analizzando le Tabelle 3 e 4 emerge subito come l'ultimo approccio abbia prestazioni effettivamente quasi costanti e, sopratutto, di un ordine di grandezza 10 volte più piccolo dell'implementazione con Flag Booleano.

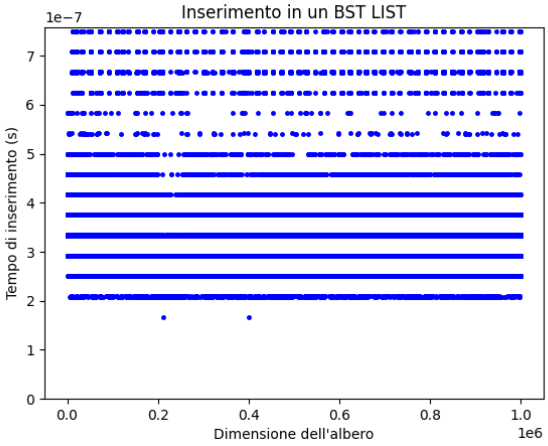

Figura 12: Prestazioni di inserimento di 1.000.000 chiavi identiche un BST con Lista Concatenata

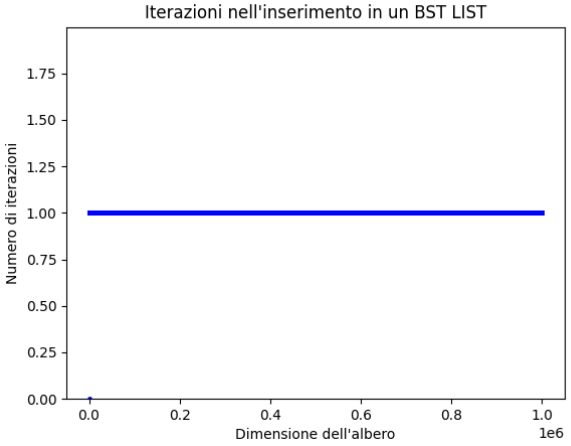

Figura 13: Iterazioni nell'inserimento di 1.000.000 chiavi identiche un BST con Lista concatenata

Numero di elementi	Tempo di esecuzione (s)
0	2.46e-06
76920	2.13e-06
153840	2.13e-06
230760	2.75e-06
307680	3.00e-06
384600	3.13e-06
461520	3.08e-06
538440	3.08e-06
615360	7.46e-06
692280	2.33e-06
769200	3.08e-06
846120	3.17e-06
923040	3.08e-06
999960	3.17e-06
999990	3.12e-06

Tabella 3: Inserimento in un BST con Flag Booleano

Numero di elementi	Tempo di esecuzione (s)
0	4.75e-06
76920	2.92e-07
153840	2.50e-07
230760	2.50e-07
307680	2.50e-07
384600	2.50e-07
461520	3.33e-07
538440	3.75e-07
615360	3.34e-07
692280	3.33e-07
769200	2.92e-07
846120	3.33e-07
923040	3.75e-07
999960	5.00e-07
999990	3.33e-07

Tabella 4: Inserimento in un BST con Lista Concatenata

Vediamo infine ciò che è veramente significativo: la visualizzazione del confronto diretto tra i 3 approcci. Risulta quindi subito evidente come entrambi i due approcci della lista concatenata e del flag booleano, nonostante magari la necessità di una iniziale progettazione e implementazione più specifica, portano enormi benefici. Entrambi i grafici di Figura 14 e Figura 15 sono stati realizzati sulla base degli stessi dati già mostrati; il secondo è però più zoomato sull'asse delle y così che si possano apprezzare anche le misurazioni di *BST FLAG* e *BST LIST*, prticamente impercettibile nella figura Figura 14 dominata dal grafico del primo approccio.

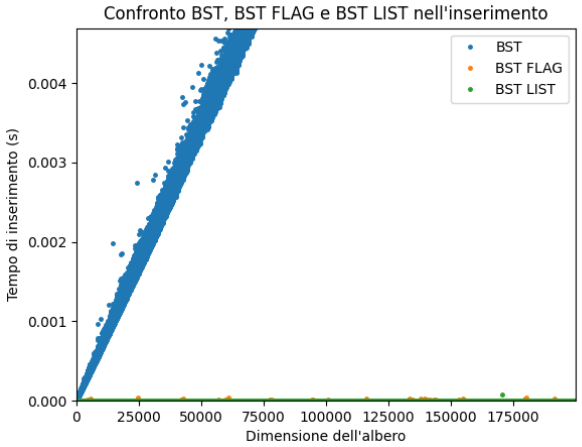

Figura 14: Confronto tra le prestazioni di inserimento dei 3 tipi di BST

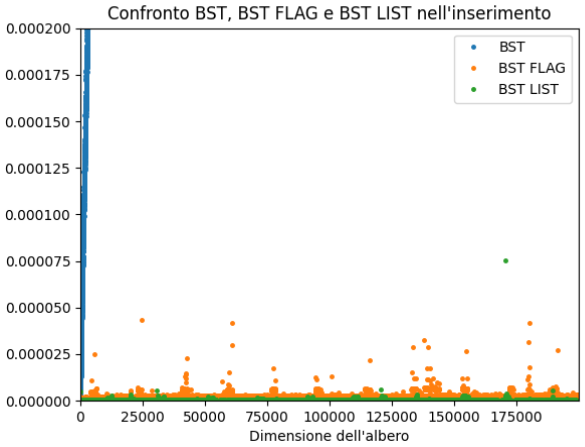

Figura 15: Zoom del confronto tra le prestazioni di inserimento dei 3 tipi di BST

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.