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

Nel contesto della struttura dati degli Alberi Binari di Ricerca, la presente relazione si propone di esaminare e confrontare diverse strategie implementative al fine di valutarne l'efficacia e l'impatto sulle prestazioni. Gli Alberi Binari di Ricerca costituiscono una struttura fondamentale in molte applicazioni *"real word"* ma, come vedremo in seguito, la gestione delle chiavi duplicate può essere complessa. Questa breve analisi si concentrerà su tre distinti approcci, non prima però di aver esposto brevemente il concetto di Albero Binario senza chiavi duplicate. Saranno poi analizzati dei test di prestazioni sia dal punto di vista teorico, sia dal punto di vista empirico, per poi giungere alle conclusioni. La relazione è accompagnata da estratti di codice, tabelle riassuntive e grafici, e si invita inoltre a consultare i risultati allegati. Infine, per chiarimenti sulle specifiche dei test si rimanda alla nota dell'ultimo paragrafo, seguito poi dalla bibliografia.

2 Nozioni sugli Alberi Binari di Ricerca

Prima di esaminare il problema delle chiavi duplicate cerchiamo di capire cos'è un albero binario di ricerca (abbreviato in BST dall'inglese *Binary Search Tree*).

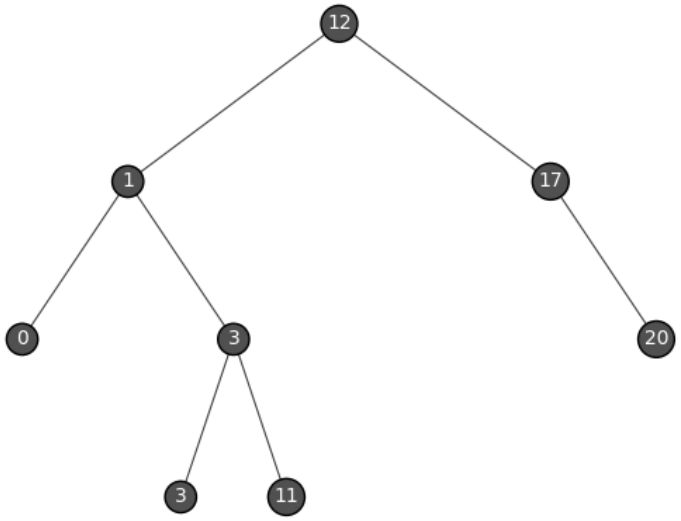


Figura 1: Esempio di albero binario di ricerca

2.1 Cos'è un Albero Binario di Ricerca?

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 l'esecuzione delle operazioni di base con complessità O(h), dove h è l'altezza dell'albero, proprietà che li 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 (e quindi classi) 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. L'albero in figura ha però ha due nodi con stessa chiave, la chiave 3! Come fare a gestire questa situazione? Sarà argomento del paragrafo 3.

2.2 Implementazione in Python

Come già accennato, in informatica 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, parent=None):
        self.key = key
        self.left = None
        self.right = None
        self.p = parent

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

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, la ricerca e tutto ciò che lo aggrada. 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 che 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 la

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 che ho usato per la rappresentazione grafica dei BST in questa relazione.

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 discussione ne verranno trattati 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 in modi diversi 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 nessun accorgimento rispetto al caso senza chiavi duplicate; tuttavia, ciò è molto limitante: sebbene per motivi didattici quando si tratta i BST si tende a semplificare considerando ogni nodo solo caratterizzato 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.

Ne approfitto quindi per includere qui la funzione di ricerca in BST senza chiavi duplicate, che verrà successivamente usata come riferimento.

Codice 3: Ricerca in un BST senza chiavi duplicate

```
def get(self, key):
    return self._getNode(self.root, key)

def _getNode(self, currentNode, key):
    if (currentNode is None): return None
    if (key == currentNode.key): return currentNode
    if (key < currentNode.key):
        return self._getNode(currentNode.left, key)
    else:
        return self._getNode(currentNode.right, key)
```

3.2 Implementazione "normale"

Un primo possibile approccio alla questione delle chiavi duplicate, e 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'operazione di inserimento nella classe BST: al momento dell'inserimento i nodi con stessa chiave vengono inseriti nel sotto-albero di sinistra del primo nodo con tale chiave (e così via ricorsivamente). Aggiustiamo quindi il codice.

Codice 4: 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)
```

In Figura 3 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 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 4: 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. Eviterò di mostrare anche il codice della ricerca, disponibile nel progetto allegato, ma questo dovrà essere un estensione del metodo già mostrato (Vedi codice 3); la differenza risiede nel fatto che, una volta trovata una chiave, bisogna rilanciare ricorsivamente sul ramo sinistro la medesima funzione. Infatti, come appena fatto notare dalla Figura 4 non siamo sicuri di aver trovato tutte le chiavi finché non arriviamo a una foglia. Creiamo quindi un array inizialmente vuoto e locale al metodo per poi passarlo come argomento alla funzione ausiliaria ricorsiva: una volta trovata una chiave duplicata verrà quindi messa nell'array.

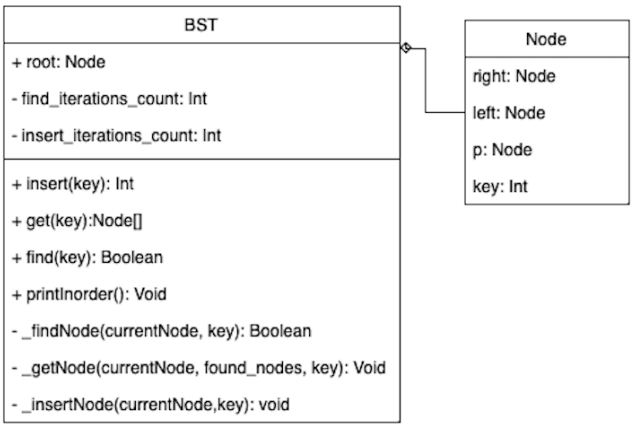


Figura 2: UML delle classi del BST

Ecco il diagramma delle classi per questa implementazione di Albero Binario. Come possibile vedere, ho deciso di implementare anche metodi di supporto e operazioni aggiuntive di cui non ho riportato il codice per motivi riassuntivi. Stesso discorso per gli attributi, ho voluto aggiungere due variabili di supporto, *find_iterations_count* e *insert_iterations_count*, per tenere traccia del numero i ricorsioni durante le operazioni, ovviamente solo al fine degli esperimenti.

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 5).

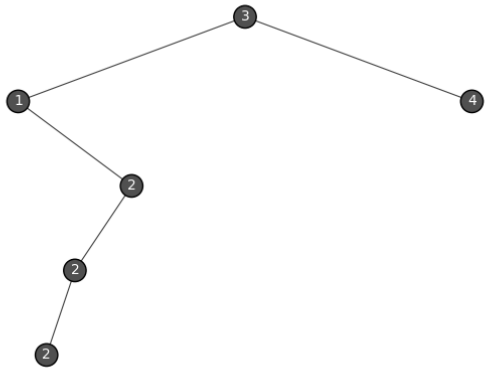


Figura 3: BST con chiavi duplicate

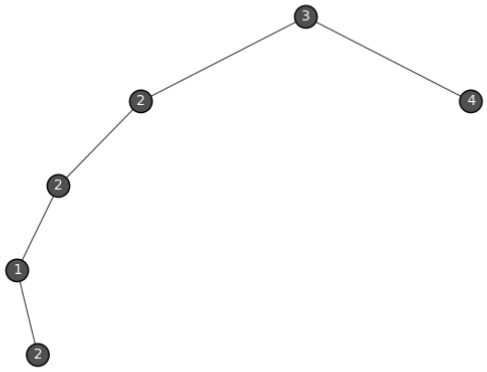


Figura 4: Altro esempio di BST con chiavi duplicate

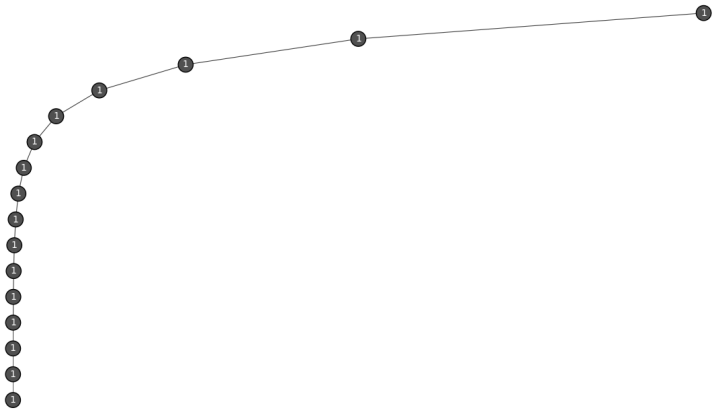


Figura 5: Esempio di albero binario di ricerca sbilanciato

3.3 Metodo del Falg Booleano

Un altro possibile approccio al caso delle chiavi duplicate si basa su invece di inserire i nodi con chiave duplicate sempre a sinistra del nodo padre, farlo a sinistra o destra del padre a seconda di un flag booleano salvato nel nodo stesso. 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 di y, poniamo y 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 6).

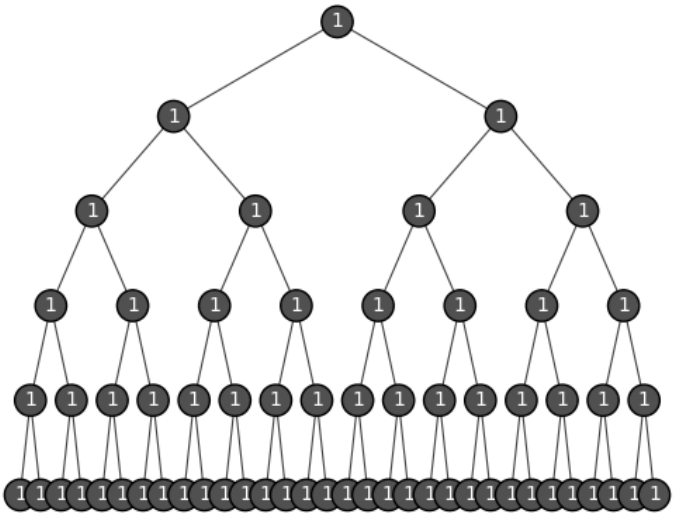


Figura 6: BST con Flag Booleano di 64 chiavi uguali

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

Codice 5: Classe Nodo di un BST con Flag Booleano

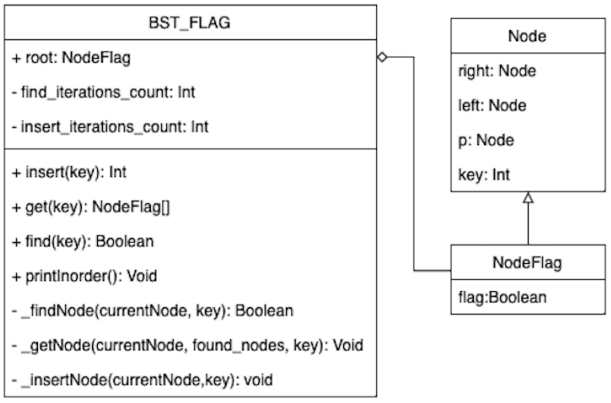
```
class NodeFlag(Node):
    def __init__(self, key, parent=None):
        super().__init__(key, parent)
        self.flag = False
```

E aggiorniamo la funzione di supporto per l'inserimento:

```
def _insertNode(self, currentNode, key):
        self.insert_iterations_count += 1

        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 = NodeFlag(key, currentNode)
```

```
# 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 = NodeFlag(key, currentNode)
```

La ricerca, invece si complica leggermente: come nel caso precedente se si trova un valore con chiave uguale si prosegue la ricerca, ma mentre nell'implementazione "normale" è sufficiente richiamare la ricorsione nel nodo a sinistra, in questo caso non abbiamo la certezza che i nodi con stessa chiave inseriti successivamente siano stati collocati a sinistra o a destra. Dobbiamo quindi proseguire la ricerca sia nel sotto-albero destro, sia nel sotto-albero sinistro.



Come evidente nel diagramma UML, ho deciso di implementare le modifiche sui nodi dell'albero estendendo la classe Nodo tramite ereditarietà. Avrei potuto fare la stessa cosa anche con la classe Albero, ma ho preferito evitare il polimorfismo di Python, non molto utilizzato.

Figura 7: UML delle classi per BST con Flag Booleano

3.4 Metodo della Lista Concatenata

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 campo *duplicates* che, nel primo nodo, tiene il conteggio dei nodi con chiave duplicata per esigenze rappresentative.

Codice 6: Classe Nodo di un BST con lista concatenata

```
class NodeList(Node):
    def __init__(self, key, parent=None):
        super().__init__(key, parent)
        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 in testa alla lista

Codice 7: Gestione chiavi duplicate con lista concatenata

```
if (key == currentNode.key):
    newNode = NodeList(key, currentNode)
    newNode.next = currentNode.next
    if(currentNode.next): currentNode.next.p = newNode
    currentNode.next = newNode
    currentNode.duplicates += 1
```

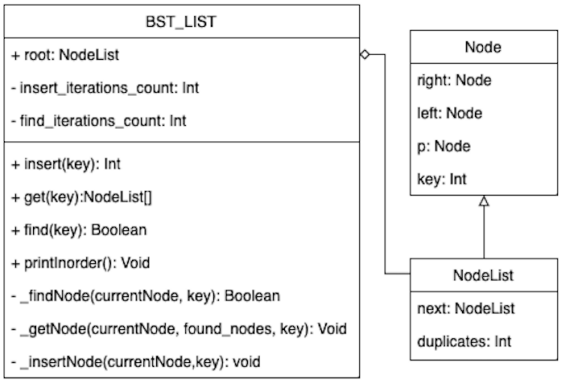


Figura 8: UML delle classi per BST con Lista Concatenata

Apportando una semplice modifica alla funzione *plot_tree* (Vedi codice 2) possiamo visualizzare questo approccio in Figura 9 e Figura 10. Per ogni nodo è segnato tra parentesi il numero di occorrenze di quella chiave.

Per quanto concerne la ricerca, in questo approccio essa si semplifica notevolmente: una volta trovato il primo nodo con la chiave voluta tramite il metodo esposto nel Codice 3, sarà sufficiente scorrere la lista concatenata per avere tutti gli altri nodi con la medesima chiave.

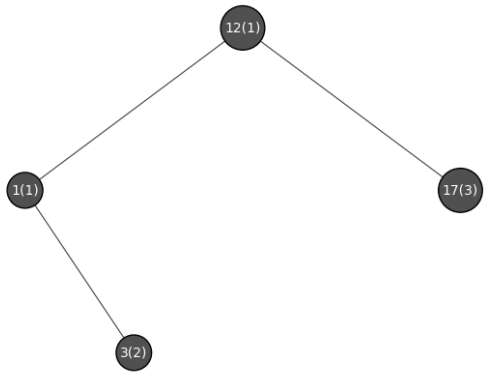


Figura 9: BST con gestione chiavi duplicate tramite lista concatenata

Figura 10: BST con lista concatenata di 100 chiavi uguali

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 dell'operazione di ricerca, con N che tende a un numero molto grande.

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 nel 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 5, 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 del Flag Booleano (Paragrafo 3.3) evita questa problematica e cerca di preservare il bilanciamento dell'albero. L'altezza diventa quindi funzione logaritmica e la complessità di l'inserimento $\Theta(log(n))$.
Infine per quanto riguarda 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 10) e quindi la complessità risultante diventa $\Theta(1) + \Theta(1) = \Theta(1)$.

4.2 Risultati Sperimentali

Come più volte ripetuto il test eseguito consiste nell'inserire N (numero molto grande) volte la stessa chiave in un albero binario di ricerca per comprendere in questo caso estremo le differenze tra le tre implementazioni. Ho strutturato il progetto in modo da evitare il più possibile la duplicazione di codice. Ho quindi creato una classe Albero per ognuno dei tre tipi e ho poi scritto una funzione di test che, preso in ingresso un'istanza di una qualsiasi classe albero, esegue i test di ricerca e inserimento su tale classe per un numero di volte passato come parametro, salvando poi i risultati sul disco in file *xlsx* e *csv*.
Il primo approccio, come ipotizzato nel paragrafo precedente, si è rilevato il più dispendioso di risorse computazionali e di tempo. Infatti, se i grafici generati per gli altri due casi arrivano 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 sino a 200.000 chiavi hanno impiegato sul mio calcolatori (Vedi la nota sui test) più di mezz'ora di esecuzione. I test con N nell'ordine dei milioni, che ho più volte tentato, erano quasi non praticabili, e a mio avviso nemmeno significativi. 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 impiegate 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 iterazioni e, sopratutto, la tabella (1) che ci mostra chiaramente l'identità tra il numero di elementi dell'albero e il numero di iterazioni necessarie per l'inserimento (questo ovviamente perché necessario scorrere tutto l'albero degenerato in una lista concatenata).

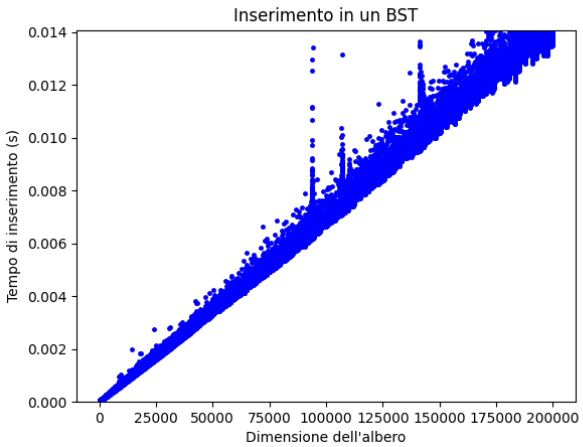


Figura 11: Prestazioni di inserimento di 200.000 chiavi identiche in un BST senza accorgimenti particolari

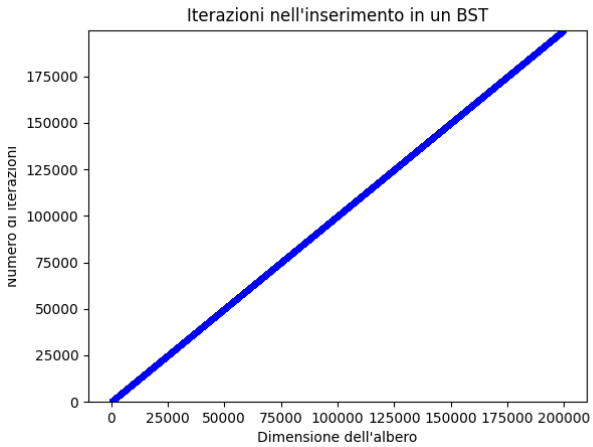


Figura 12: Variazione del numero di iterazioni per l'inserimento in un BST di 200.000 chiavi identiche

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 salvati i risultati in file *.xlsx* da migliaia di righe. Ne consegue che, per ovvi motivi, questi file sono stati successivamente abbreviati da un altro script Python selezionando solo le misurazioni più significative. Si invita comunque a consultare i risultati allegati.

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 cresca 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, mettendo le mani avanti 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 ciò, dopo aver messo in atto piccoli 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 più evidente se si prende in considerazione il grafico delle iterazioni di inserimento.

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 di interesse, come nel caso di Figura 15, poiché in questo caso l'operazione di inserimento risulta molto veloce e poco dispendiosa di risorse, accade che la variabile di incertezza/casualità influenza maggiormente il risultato. Ecco perché il grafico in Figura 15 non risulta perfettamente come costante, ma i valori dei test si distribuiscono quasi uniformemente in un intervallo. È inoltre fondamentale notare come la scala verticale dei grafici di Figura 15 e Figura 13 sia diversa; 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.

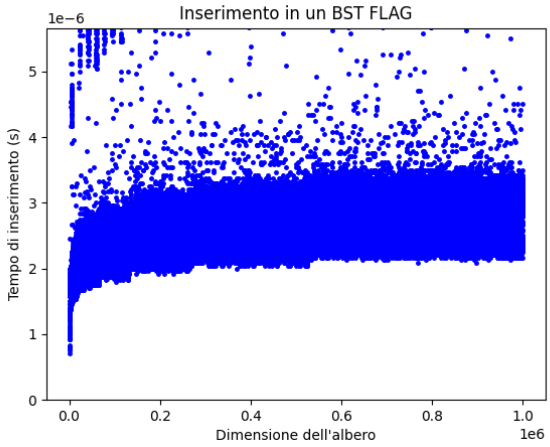


Figura 13: Prestazioni di inserimento di 1.000.000 chiavi identiche in un BST con Flag booleano

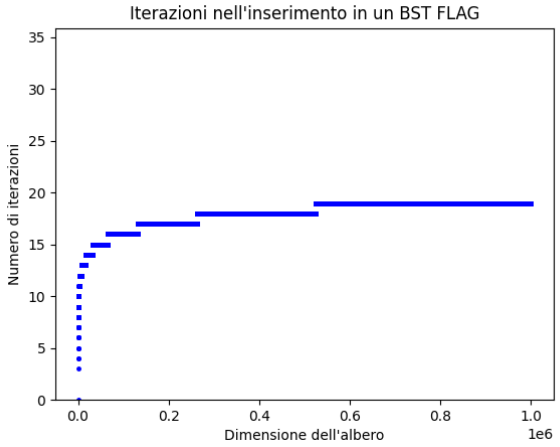


Figura 14: Ricorsioni nell'inserimento di 1.000.000 di chiavi identiche in un BST con Flag booleano

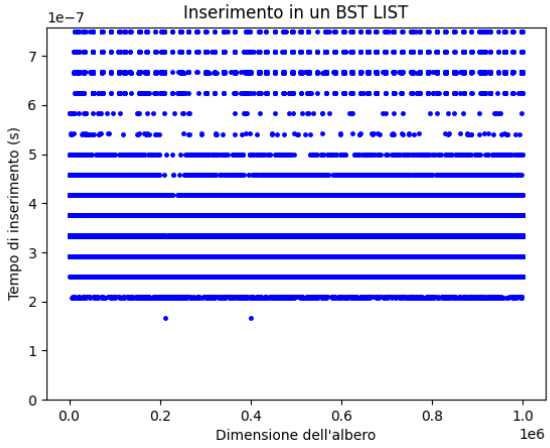


Figura 15: Prestazioni di inserimento di 1.000.000 di chiavi identiche in un BST con Lista Concatenata

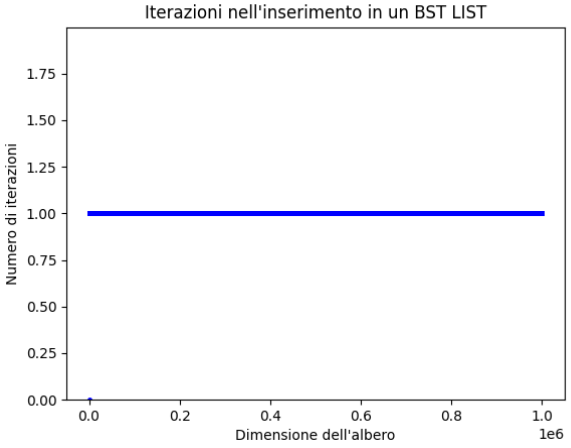


Figura 16: Ricorsioni nell'inserimento di 1.000.000 di chiavi identiche in un BST con Lista concatenata

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 subito evidente come entrambi gli 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 17 e Figura 18 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*, praticamente impercettibile nella figura Figura 17 dominata dal grafico del primo approccio.

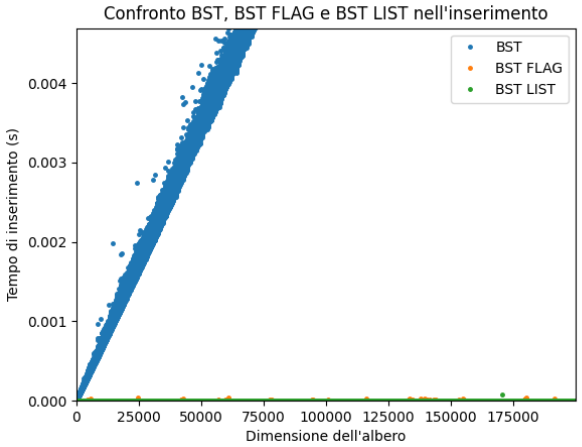


Figura 17: Confronto tra le prestazioni di inserimento dei 3 tipi di BST

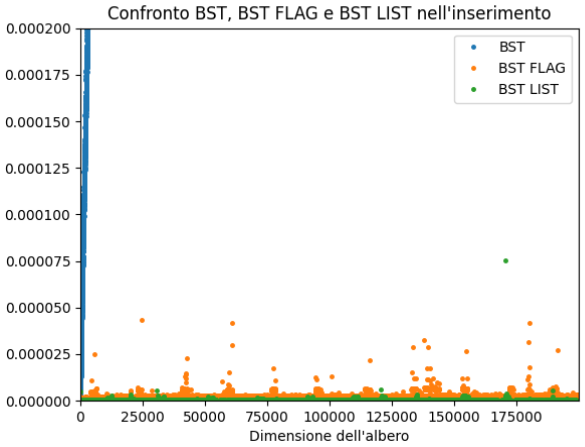


Figura 18: Zoom del confronto tra le prestazioni di inserimento dei 3 tipi di BST

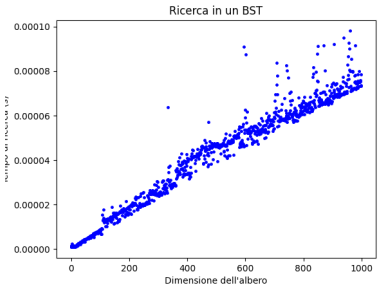


Figura 19: Prestazione di ricerca in un BST

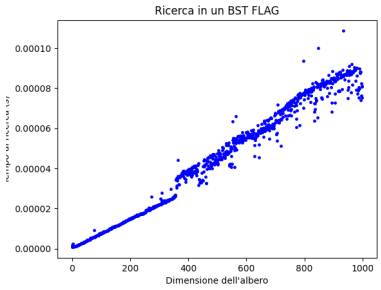


Figura 20: Prestazioni di ricerca in un BST con Flag Booleano

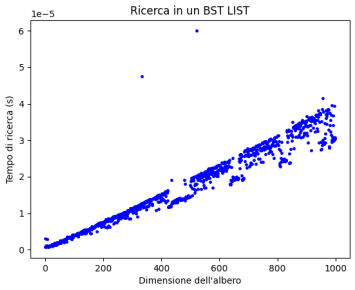


Figura 21: Prestazioni di ricerca in un BST con Lista Concatenata

Durante lo svolgimento di questi test, per ognuno dei tre tipi di BST e per gli stessi valori di n, è stata anche eseguita l'operazione di ricerca (Vedi premessa nel paragrafo 3.1). Tuttavia, come ci si poteva aspettare in questo specifico caso i risultati ottenuti non sono per nulla significativi. Ho deciso di limitare i test a soli 1000 elementi, un campione di dati significativamente ridotto ma comunque sufficiente per lo scopo. È di banale evidenza, infatti, che nel modo in cui si è intesa la ricerca, devono essere trovate tutte le occorrenze della chiave, ma essendo il test su alberi composti solo da tale chiave dovrà essere attraversato ogni nodo. Il costo della ricerca cresce quindi linearmente con l'aumentare di n, come si può evincere dai grafici riportati (Figura 19, Figura 20, Figura 21).

4.3 Cenni sulla cancellazione

Per quanto riguarda l'operazione di cancellazione, se non è stata inclusa nei test è perché ritengo che non sia significativa nell'ottica di questo studio. Brevemente: è noto come la cancellazione in un BST si articola su vari possibili casi; se il nodo da cancellare non ha figli o solo un figlio sarà sufficiente eliminare il nodo e trasferire il figlio al nodo padre. Le cose si complicano se il nodo da eliminare ha sia figlio destro, sia figlio sinistro: non entrerò nel dettaglio, ma è necessario eseguire una o due operazioni di trapianto a seconda se il successore del nodo è l'immediato figlio destro o no. Ciò che conta è che l'operazione ha un costo di $O(h)$ senza però contare la ricerca del nodo da cancellare. Quello che cambia tra i tre alberi è quindi la ricerca, già trattata, e il fatto che se nei primi due approcci dopo aver eliminato il primo nodo bisogna continuare la ricerca per eliminare anche gli altri, il terzo approccio ha il vantaggio che per cancellare tutti i nodi con la stessa chiave è sufficiente eliminare solo il primo di nodo.

5 Conclusioni

È giunto il momento delle conclusioni. Tengo a sottolineare nuovamente come i test eseguiti rappresentino il caso peggiore e quindi un caso di utilizzo ben oltre l'estremo che in situazione reali difficilmente si presenta. Sarebbe stato opportuno generare quindi un dataset meno sbilanciato in termini di chiavi duplicate, ma non avrebbe reso con tale chiarezza le differenze tra i tre alberi. Il test ha reso chiaro come se il primo approccio è immediato e semplice nella sua implementazione, questo non è per nulla efficiente nella gestione delle chiavi duplicate se in grande numero. Il secondo approccio migliora notevolmente il bilanciamento dell'albero nel caso di molte chiavi duplicate e quindi di conseguenza le prestazioni, non più lineari ma logaritmiche. La sua pecca però consiste in un'implementazione che complica ulteriormente la ricerca. Infine, l'ultimo approccio è quello che sicuramente ha avuto prestazioni migliori durate i test, persino lineari! Non è però così tutto semplice, poiché questo risultato è legato esclusivamente alla natura del test eseguito. Ciò comunque non esclude che tale approccio, a fronte di un'implementazione più onerosa in termini di accorgimenti necessari, possa offrire effettivamente ottime prestazioni anche i casi di uso reali, stando anche qui ovviamente attenti a non far degenerare l'albero in una semplice lista. Per finire quindi, la scelta delle implementazioni ovviamente non è immediata, ma va decisa sulla base di vari fattori tra cui il tipo di operazioni che interessa eseguire sulla struttura dati e sopratutto la quantità di elementi con chiave duplicata che si stima di avere.

Nota sui test

Si tenga presente che per 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 questa 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.