UNIVERSITÀ DEGLI STUDI FIRENZE

UNIVERSITÀ DEGLI STUDI DI FIRENZE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Laboratorio di Algoritmi e Strutture Dati

Autore:
Pellicci Simone

N° Matricola:
7111840

Corso principale:
Algoritmi e Strutture Dati

Docente corso:
Marinai Simone

Indice

1 Introduzione generale

1.1 Progetti assegnati

Qui di seguito verranno elencati i progetti che mi sono stati assegnati per il superamento della parte di Laboratorio di Algoritmi e Strutture Dati.

Confrontare le varie implementazioni di dizionario senza usare le librerie delle strutture dati di Python tramite:

- Hash
- Abr
- Lista concatenata

La descrizione dei vari esercizi è all'inizio di ognuna delle parti della ricerca.

1.2 Breve descrizione dello svolgimento degli esercizi

Per ogni esercizio suddivideremo la sua descrizione in 4 parti fondamentali:

- **Spiegazione teorica del problema** : qui è dove si descrive il problema che andremo ad affrontare in modo teorico partendo dagli assunti del libro di Algoritmi e Strutture Dati e da altre fonti.
- **Documentazione del codice** : in questa parte spieghiamo come il codice dell'esercizio viene implementato
- **Descrizione degli esperimenti condotti** : partendo dal codice ed effettuando misurazioni varie cerchiamo di verificare le ipotesi teoriche
- **Analisi dei risultati sperimentali** : dopo aver svolto i vari esperimenti riflettiamo sui vari risultati ed esponiamo una tesi

1.3 Specifiche della piattaforma di test

La piattaforma di test sarà la stessa per ogni esercizio che vedremo. Partiamo dall'hardware del computer fondamentale da conoscere per questo esercizio:

- **CPU** : AMD Ryzen 5 3550H with Radeon Vega Mobile Gfx 2.10 GHz
- **RAM** : Crucial Ballistix 16GB DDR4 3600MHz
- **SSD** : 16 GB DDR4-2400 MHz RAM (2 x 8 GB)
- **Disco di memoria** : 512 GB PCIe® NVMe™ M.2 SSD

Il linguaggio di programmazione utilizzato sarà Python 3.12.6 e la piattaforma in cui il codice è stato caricato è l'IDE **PyCharm Edu 2021.3.1**. La stesura di questo testo è avvenuta tramite l'utilizzo dell'editor online **Overleaf**.

2 Spiegazione teorica dei problemi

Il confronto delle varie implementazioni di dizionario può essere fatto tramite python dove andremo a creare tre classi chiamate come il tipo di implementazione, quindi: Hash, Albero binario di ricerca(ABR) e Lista concatenata(Lc). Succesivamente andreamo a confrontare i vari metodi di queste classi tramite dei grafici con un apposita libreria. Quindi i passaggi principali saranno:

- Scivere il codice per ogni tipo di implentazione
- Assicurarci che il codice compili
- Implementare delle parti di codice per fare e salvare i test
- Fare i test
- Confrontare i vari risultati

2.1 Introduzione

In questa parte andremo a spiegare il funzionamento dei vari metodi e delle varie strutture dati che andremo ad utilizzare tenendo conto dell'efficienza e le ipotesi su di essa.

2.2 Osservazioni importanti

Andremo ad analizzare tutte le strutture dati concentrandosi sulle varie casistiche che si possono presentare a seconda delle struttura dati utilizzata, quindi dovremo analizzare tre casi in particolare:

- Caso migliore
- Caso medio
- Caso peggiore

in questo modo avremo un analisi completa ed easustiva del comportamento dell algoritmo in ogni caso possibile e potremo determinare quali siano i vari pro e contro di ognuno di essi per poi decidere quale utilizzare in certe situazioni a seconda delle esigenze.

C'è un osservazione molto importante da fare sul caso migliore, cioè che in ogni caso il tempo di esecuzione è di $O(1)$ e quindi possiamo non analizzare questo caso, per motivi che verrano spiegati successivamente.

2.3 Differenze tra le varie strutture dati

La parte comune di ogni struttura dati che andremo ad analizzare sarà il dizionario che andremo a chiamare nodo, il quale sarà presente in ognuna di esse con un chiave, valore e alcuni attributi differenti che verrano spiegati successivamente.

2.3.1 Hash

La funzione hash è una funzione molto importante in informatica ma può creare conflitti, infatti ci sono vari modi per risolverli, nel corso di Algoritmi e Strutture dati ne abbiamo visti due: **Chaining** e **Indirizzamento Aperto**, noi andremo ad utilizzare in primo metodo; questo struttura dati ha le seguenti caratteristiche:

- Funzione Hash: tramite la quale si va a decidere a quale lista il nodo verrà: concatenato, recuperato o rimosso.
- Grandezza tabella: che si utilizza nella funzione ahash quando si deve calcolare l'indice; se questo attributo non è sufficientemnte grande la struttura dati si comporterà come una normale lista concatenata.

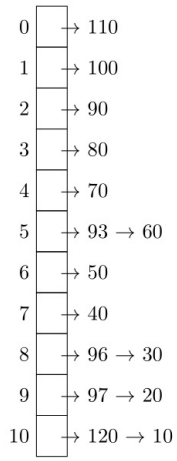

Figura 1, rappresentazione grafica del chaining

2.3.2 Albero binario di ricerca

Lo chiameremo Abr nei vari test per semplificare la scrittura del codice. Questa struttura dati ha le seguenti caratteristiche:

- I nodi: ogni singolo nodo è composto dal dizionario ed ha un massimo di 2 figli;
- Il bilanciamento: c'è il rischio che l'abero sia del tutto sbilanciato e quindi si comporterà come una lista concatenata.

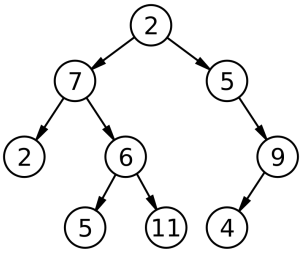

Figura 2, rappresentazione grafica di un albero binario di ricerca

2.3.3 Liste concatenate

sono chiamate anche liste a puntatori perchè sono degli Struct con all interno l indirizzo dell elemento successivo alla lista ed i vari valori, nel nostro caso gli elementi del dizionario più il puntatore; questa struttura dati ha le seguenti caratteristiche:

- Puntatori: sono degli operatori molto usati delle strutture dati ma si tende a non utilizzarli in prima persona essendo a volte complicati da utilizzare;
- Tempo di esecuzione: per ognuno dei 3 metodi implementati non si può sapere il tempo effettivo di esecuzione e neanche iportizzarlo nel caso medio dato che dipende dalla lista.

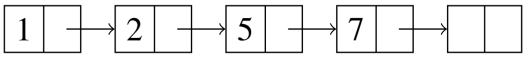

Figura 3, rappresentazione grafica di una lista concatenata

2.4 Ipotesi

Andremo ad analizzare tutti i metodi avendo implementato il dizionario nelle varie strutture dati. I tre metodi da analizzare sono l'Aggiunta di un nodo, il recupero di un nodo e la rimozione di un nodo.
Possiamo dividere in tre casi principali, senza analizzare il caso migliore come annotato precedentemente:

Caso migliore			
	Aggiungere	Recuperare	Rimuovere
Lc	$O(1)$	$O(1)$	$O(1)$
Hash	$O(1)$	$O(1)$	$O(1)$
ABR	$O(1)$	$O(1)$	$O(1)$

Caso medio			
	Aggiungere	Recuperare	Rimuovere
Hash	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
ABR	$O(log(n))$	$O(log(n))$	$O(log(n))$

Caso peggiore			
	Aggiungere	Recuperare	Rimuovere
Hash	$O(n)$	$O(n)$	$O(n)$
ABR	$O(n)$	$O(n)$	$O(n)$
Lista concatenata	$O(n)$	$O(n)$	$O(n)$

Non vi è la presenza delle liste concatenate nella tabella dato che non c'è un caso medio, dipende tutto dalla posizione della chiave che può essere ovunque nell'intervallo che va da **0** a **n-1**.

3 Documentazione del codice

3.1 Diagrammi UML e relazioni tra classi

In ogni struttura dati abbiamo una tipologia diversa di nodo, però tutte derivano da quello della lista concatenata che ha 3 attributi:

- Chiave
- Valore
- Puntatore next

Nel caso dell albero binario abbiamo un nodo con 2 puntatori next chiamati figli(destro e sinistro). Ci sono 3 tipologie di nodi in un albero: Radice, Nodo base che non ha un padre ed un massimo di 2 figli, Ramo, Nodo che ha un padre ed un massimo di 2 figli, Foglia, Nodo che ha un padre e 0 figli.

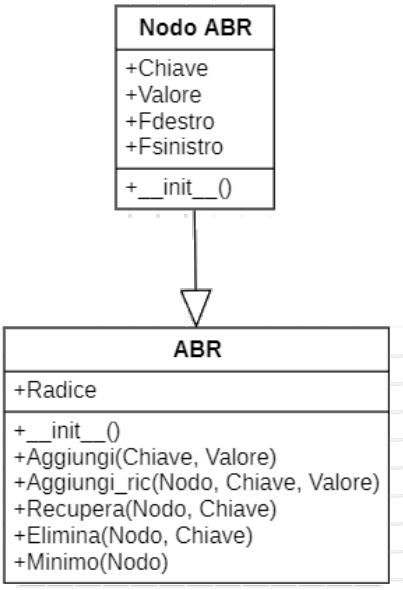

Figura 4, Grafico UML del albero binario di ricerca

Nella tabella Hash i nodi si comportano come un una lista concatenata. L'unica differenza è la presenza di una tabella che contiene più liste concantenate.

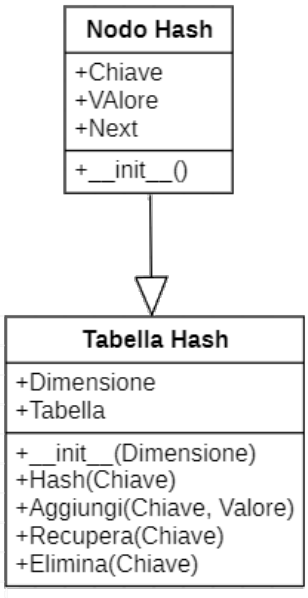

Figura 5, Grafico UML della lista concatenata

Una lista concatenata ha come unica caratteristica di avere una testa, che si può paragonare alla radice di un albero, però i nodi successivi hanno un solo figlio.

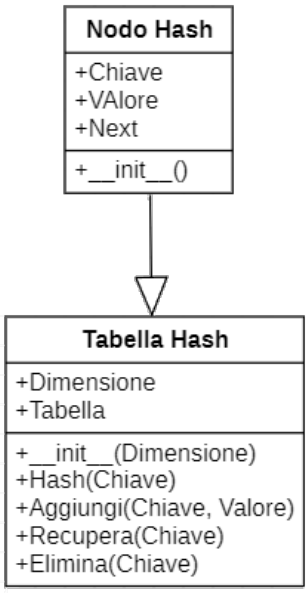

Figura 6, Grafico UML della tabella hash

3.2 Attributi e metodi delle classi

In questa sezione andremo ad analizzare il codice scritto in python delle varie strutture dati focalizzandoci sui metodi implementati ed attributi ed i corrispettivi attributi all'interno delle classi utilizzate.

3.2.1 Lista concatenata

La classe ListaConcatenata implementa una lista con solo l'attributo testa dove verrano poi aggiungti dei nodi. La classe Nodo rappresenta un nodo della lista.

- Costruttore Nodo: costruttore che inizializza la chiave, il valore e imposta il riferimento al nodo successivo (next) su None.

```
def __init__(self, chiave, valore):
    self.chiave = chiave
    self.valore = valore
    self.next = None
```

- Costruttore Lista: costruttore che inizializza la lista concatenata tramite l'attributo testa inizializzato a None

```
def __init__(self):
        self.testa = None
```

- Metodo aggiungi: aggiunge un nuovo nodo con una specifica chiave e valore. Se la lista è vuota il nuovo nodo diventa la testa altrimenti scorre tutta la lista ed aggiunge il nuovo nodo alla fine della lista.

```
def aggiungi(self, chiave, valore):
        nodo = Nodo(chiave, valore)
        if self.testa is None:
            self.testa = nodo
            return
        indice = self.testa
        while indice.next is not None:
            indice = indice.next
        indice.next = nodo
```

- Metodo recupera: cerca e restituisce il valore associato alla chiave data scorrendo i nodi della lista finché trova un nodo con la chiave cercata o raggiunge la fine. Se trova il nodo, restituisce il valore altrimenti ritorna None.

```
    def recupera(self, chiave):
        indice = self.testa
        while indice.next is not None and
        indice.chiave is not chiave:
            indice = indice.next
        if indice.next is None:
            return None
        else:
            return indice.valore
```

Metodo elimina: elimina il nodo con la chiave data. Controlla se la lista è vuota o ha un solo elemento e gestisce i casi di conseguenza altrimenti, scorre la lista per trovare ed eliminare il nodo con la chiave cercata aggiornando i puntatori dei nodi adiacenti.

```
def elimina(self, chiave):
        indice = self.testa
        nodo = None
        if indice is None:
            return False
        if indice.next is None:
            self.testa = None
            return True
        nodo = indice.next
        while nodo.next is not None and
        nodo.chiave is not chiave:
            indice = indice.next
            nodo = nodo.next
        indice.next = nodo.next
        return True
```

3.2.2 classe Albero binario di ricerca

La classe AlberoBinarioRicerca implementa un albero binario di ricerca composta solo dalla radice con valore None. La classe Nodo rappresenta un nodo dell'albero binario.

- Costruttore Nodo: costruttore che inizializza il nodo con chiave e valore, impostando i figli su None.

```
def __init__(self, chiave, valore):
    self.chiave = chiave
    self.valore = valore
    self.destra = None
    self.sinistra = None
```

- Costruttore Nodo: costruttore che inizializza la radice dell'albero su None.

```
def __init__(self):
        self.radice = None
```

- Metodo aggiungi: aggiunge un nuovo nodo con chiave e valore specificati. Se l'albero è vuoto imposta il nuovo nodo come radice altrimenti chiama aggiungi-ric per inserire il nodo in posizione corretta.
- Metodo aggiungi-ric: è funzione ricorsiva per trovare la posizione di un nuovo nodo, se la chiave è minore di quella di nodo cerca o inserisce a sinistra altrimenti se la chiave è maggiore cerca o inserisce a destra.

```
def aggiungi(self, chiave, valore):
        if self.radice is None:
            self.radice = Nodo(chiave, valore)
        else:
            self._aggiungi_ric(self.radice, chiave, valore)

def _aggiungi_ric(self, nodo, chiave, valore):
    if chiave < nodo.chiave:
        if nodo.sinistra is None:
            nodo.sinistra = Nodo(chiave, valore)
        else:
            self._aggiungi_ric(nodo.sinistra, chiave, valore)
    elif chiave > nodo.chiave:
        if nodo.destra is None:
            nodo.destra = Nodo(chiave, valore)
        else:
            self._aggiungi_ric(nodo.destra, chiave, valore)
```

- Metodo recupera: cerca e restituisce il valore associato a chiave tramite la funzione ricorsiva recupera-ric.
- Metodo recupera-ric: funzione ricorsiva che si sposta nel figlio sinistro se la chiave data è maggiore del nodo in cui ci troviamo e fa il contrario nel caso in cui la chiave sia minore. Ritorna True se è presente la chiave nell'albero sennò False.

```
def recupera(self, chiave):
        return self.recupera_ric(self.radice, chiave)

def recupera_ric(self, nodo, chiave):
    if nodo is None:
        return None
    if chiave == nodo.chiave:
        return nodo.valore
    elif chiave < nodo.chiave:
        return self.recupera_ric(nodo.sinistra, chiave)
    else:
        return self.recupera_ric(nodo.destra, chiave)
```

- Metodo elimina: rimuove il nodo con la chiave specificata chiamando la funzione elimina-ric.
- Metodo elimina-ric: funzione ricorsiva per eliminare il nodo con la chiave specificata. Se il nodo ha solo un figlio, sostituisce il nodo con il figlio. Se il nodo ha due figli, trova il successore in ordine (quindi il minimo del sottoalbero destro)per sostituirlo con il nodo da eliminare, successivamente il nodo trovato con la funzione di minimo viene rimosso dalla vecchia posizione in modo da non avere duplicati.

```
def elimina(self, chiave):
        self.radice = self.elimina_ric(self.radice, chiave)

def elimina_ric(self, nodo, chiave):
    if nodo is None:
        return nodo
    if chiave < nodo.chiave:
        nodo.sinistra = self.elimina_ric(nodo.sinistra, chiave)
    elif chiave > nodo.chiave:
        nodo.destra = self.elimina_ric(nodo.destra, chiave)
    else:
        if nodo.sinistra is None:
            return nodo.destra
        if nodo.destra is None:
            return nodo.sinistra
        temp = self._minimo(nodo.destra)
        nodo.chiave = temp.chiave
        nodo.valore = temp.valore
        nodo.destra = self.elimina_ric(nodo.destra, temp.chiave)
    return nodo
```

- Metodo minimo: restituisce il nodo con la chiave più piccola nel sottoalbero.

```
def _minimo(self, nodo):
        corrente = nodo
        while corrente.sinistra is not None:
            corrente = corrente.sinistra
        return corrente
```

3.2.3 Tabella Hash

La classe TabellaHash implementa la struttura della tabella hash con una lista di dimensione fissa. La classe Nodo rappresenta un nodo singolo per la gestione delle collisioni nella tabella hash (usando il chaining).

- Costruttore Nodo: costruttore che inizializza il nodo con chiave, valore e imposta prossimo a None.

```
def __init__(self, chiave, valore):
    self.chiave = chiave
    self.valore = valore
    self.prossimo = None
```

- Costruttore Tabella Hash: costruttore che inizializza una tabella hash con la dimensione specificata.

```
def __init__(self, dimensione):
        self.dimensione = dimensione
        self.tabella = [None] * dimensione
```

- Funzione hash: calcola l'indice della tabella usando la funzione hash incorporata di Python e il modulo dimensione.

```
def hash(self, chiave):
        return hash(chiave) % self.dimensione
```

- Metodo aggiungi: aggiunge un elemento alla tabella hash: Calcola l'indice tramite la funzione hash. Se il nodo è None, inserisce il nuovo nodo altrimenti, attraversa la lista collegata per risolvere le collisioni fino in fondo alla lista ed aggiunge un nodo.

```
def aggiungi(self, chiave, valore):
        indice = self.hash(chiave)
        nodo = self.tabella[indice]

        if nodo is None:
            self.tabella[indice] = Nodo(chiave, valore)
            return

        while nodo is not None:
            if nodo.chiave == chiave:
                nodo.valore = valore
                return
            if nodo.prossimo is None:
                nodo.prossimo = Nodo(chiave, valore)
                return
            nodo = nodo.prossimo
```

- Metodo recupera: recupera il valore associato a una chiave tramite l'indice trovato con la funzione hash. Scorre la lista collegata per trovare la chiave corrispondente e restituirne il valore. Se la chiave non è presente, restituisce None.

```
def recupera(self, chiave):
        indice = self.hash(chiave)
        nodo = self.tabella[indice]

        while nodo is not None:
            if nodo.chiave == chiave:
                return nodo.valore
            nodo = nodo.prossimo

        return None
```

- Metodo elimina: rimuove un nodo dato calcolando l'indice tramite la funzione hash e poi scorre la lista collegata finchè non trova la chiave, se esiste elimina ed aggiorna la lista altrimenti ritorna False.

```
def elimina(self, chiave):
        indice = self.hash(chiave)
        nodo = self.tabella[indice]
        precedente = None

        while nodo is not None:
            if nodo.chiave == chiave:
                if precedente is None:
                    self.tabella[indice] = nodo.prossimo
                else:
                    precedente.prossimo = nodo.prossimo
                return True
            precedente = nodo
            nodo = nodo.prossimo

        return False
```

4 Descrizione dei test svolti

4.1 Casistiche analizzate

Nei test svolti, abbiamo analizzato le prestazioni delle operazioni principali delle strutture dati: aggiunta, recupero ed eliminazione dei nodi. Ogni operazione è stata misurata in due casistiche specifiche:

Caso Ordinato (Worst Case): In questo scenario, le chiavi sono aggiunte in ordine crescente, rappresentando una situazione in cui ogni operazione richiede una scansione più lunga della lista, causando tempi di esecuzione progressivamente crescenti.

Caso Casuale (Average Case): Per simulare un utilizzo realistico, le chiavi sono state mescolate casualmente prima di essere utilizzate.

4.2 Dataset

Il dataset utilizzato per i test è composto da una sequenza di chiavi numeriche intere, che vanno da 1 fino alla dimensione data in entrata al test. Per semplificare il codice possiamo associare ad ogni indice il numero stesso che lo rappresenta dato lavoreremo solo sugli indici essendo un dizionario, quindi disinteressandoci completamente del valore associato alla chiave permettendoci così di poter assegnargli un qualsiasi valore.

Per ogni esperimento, le chiavi sono state ordinate in due modalità: in sequenza crescente per testare il comportamento nel caso peggiore, in ordine casuale per misurare i tempi medi delle operazioni in condizioni variabili.
Quindi utilizzeremo il seguente codice utilizzando la libreria random nel caso in cui avremo la variabile bool: rand = True.

```
import random

chiavi = list(range(1, dim + 1))
if rand:
    random.shuffle(chiavi)
```

4.3 Codice test

Per misurare i vari tempi di esecuzione dei metodi in cui abbiamo implementato nelle varie strutture dati abbiamo bisogno di utilizzare una libreria che ci permetta di misurare il tempo:

```
import time
```

Per ogni struttura dati sono stati implementati nel main i seguenti metodi per poter misurare rispettivamente i tempi di esecuzione.

- Misura tempo aggiunta:

```
def misura_tempo_aggiungi(lista, chiavi):
    tempi = []
    for chiave in chiavi:
        inizio = time.perf_counter()
        strutturaDati.aggiungi(chiave, chiave)
        fine = time.perf_counter()
        tempi.append(fine - inizio)
    return tempi
```

- Misura tempo recupera:

```
def misura_tempo_recupera(lista, chiavi):
    tempi = []
    for chiave in chiavi:
        inizio = time.perf_counter()
        strutturaDati.recupera(chiave)
        fine = time.perf_counter()
        tempi.append(fine - inizio)
    return tempi
```

- Misura tempo elimina:

```
def misura_tempo_elimina(Hash, chiavi):
    tempi = []
    chiavi = list(reversed(chiavi))
    for chiave in chiavi:
        inizio = time.perf_counter()
        strutturaDati.elimina(chiave)
        fine = time.perf_counter()
        tempi.append(fine - inizio)
    return tempi
```

Nell ultimo caso abbiamo modificato il metodo per misurare i tempi del metodo elimina in tutte le strutture dati, questo perchè se lo implementassimo normalmente come gli altri andremo ad eliminare sempre l'elemento in testa alla lista, o radice a seconda della struttura dati, dato che passiamo le chiavi nello stesso ordine del metodo aggiungi.

4.3.1 Stampa e registrazione dei dati

L'ultima parte del codice utilizzato nei test riguarda la stampa e registrazione dei dati acquisiti.

Per migliorare la qualità dei dati e cercare di limitare dei risultati "anomali" dovuti al compilatore, ad esempio quando deve accedere a delle zone di memoria differenti dove sono registrati alcuni dati, possiamo "smussare" i dati così da avere meno curve.

Un modo per "smussare" i dati è la media mobile anche chiamata convoluzione tramite la libreria numpy, questa funzione ha in entrata i dati e li arrotonda in base alla finestra.

```
import numpy as np

def media_mobile(dati, finestra):
    kernel = np.ones(finestra) / finestra
    dati_smussati = np.convolve(dati, kernel, mode='same')
    return dati_smussati
```

Per raccogliere i vari dati dei test come: tempi di aggiunta, tempi di recuper, tempi di eliminazione ed il numero dei nodi su cui si è svolto il test; Si è utilizza la libreria pandas in modo da poter creare una tabella adatta.

```
import pandas as pd

dati_tabella = {
    'Nodi': list(range(1, dimensione + 1)),
    'Tempo Aggiungi (s)': tempi_aggiungi,
    'Tempo Recupera (s)': tempi_recupera,
    'Tempo Elimina (s)': tempi_elimina
}
df = pd.DataFrame(dati_tabella)
```

Per la stampa dei risultati si utilizza la libreria matplotlib, che ci permette di creare e manipolare i dati forniti in modo tale da vedere l'andatura dei tempi in base al numero dei nodi.

per visualizzare meglio le differenze tra le varie strutture dati si può separare le varie curve e metterle in scala differente se vogliamo visualizzarle al meglio, questo è il caso dell'ABR nel caso medio, dove infatti abbiamo messo una scala logaritmica.

```
import matplotlib.pyplot as plt

plt.figure(figsize=(12, 8))

plt.subplot(3, 1, 1)
plt.plot(df['Nodi'], df['Tempo Aggiungi (s)'],
label="Tempo di aggiunta (ABR)", color="blue")
plt.title("Tempo di esecuzione per aggiunta (ABR)")
plt.xlabel("Numero di nodi")
plt.ylabel("Tempo di esecuzione (secondi)")
if rand == True:
    plt.yscale("log")
plt.grid(False)

plt.show()
```

5 Analisi risultati

Le avvrebiazione WC e AC significano Worst Case e Avarage Case.

5.1 Lista

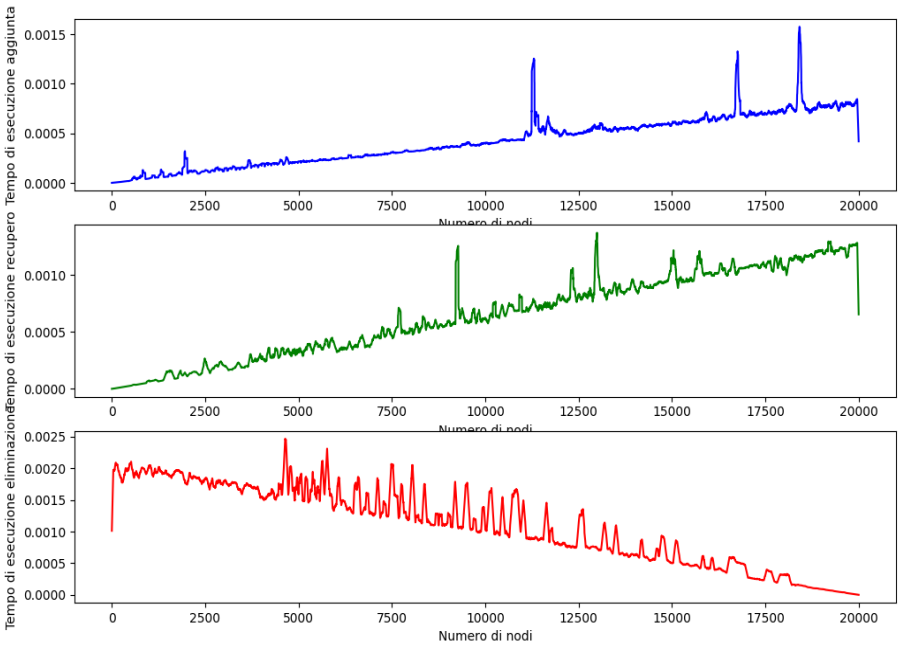

Figura 7, Grafico tempi di esecuzione di ogni nodo WC

Possiamo notare che sono presenti molte curve ma l'andatura generale è quella che ci aspettavamo, cioè $O(n)$.

Tabella 1: Tempi di esecuzione nel caso peggiore (WC)

Numero Nodi	Aggiunta (s)	Recupero (s)	Eliminazione (s)
1	0.000001	8.506669e-07	0.001013
2	0.000001	8.840005e-07	0.001036
3	0.000001	9.173333e-07	0.001059
4	0.000001	9.506669e-07	0.001081
5	0.000001	9.853331e-07	0.001104
...	...	...	...
2499	0.000121	0.0002501	0.001826
2500	0.000124	0.0002499	0.001826
2501	0.000124	0.0002496	0.001832
...	...	...	...
4999	0.000216	0.0003474	0.001586
5000	0.000216	0.0003481	0.001587
5001	0.000216	0.0003479	0.001595
...	...	...	...
9999	0.000402	0.0006261	0.001057
10000	0.000402	0.0006255	0.001043
10001	0.000402	0.0006244	0.001037
...	...	...	...
19999	0.000448	0.0007050	0.000001
20000	0.000440	0.0006886	0.000001
20001	0.000433	0.0006721	0.000001

5.2 ABR

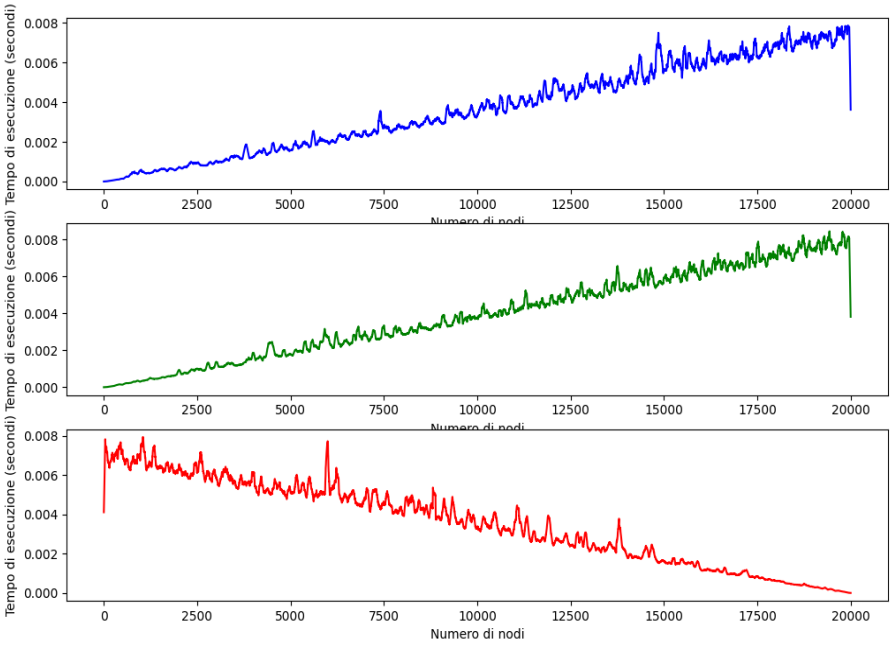

Figura 8, Grafico tempi di esecuzione di ogni nodo WC

L'andatura è come quella che ci aspettavamo nella spiegazione teorica del problema, cioè come quella di una normale lista concatenata $O(n)$ nel caso peggiore.

Tabella 2: Tempi di esecuzione nel caso peggiore (WC)

Numero Nodi	Aggiunta (s)	Recupero (s)	Eliminazione (s)
1	0.000002	0.000002	0.004115
2	0.000002	0.000002	0.004247
3	0.000002	0.000002	0.004340
4	0.000002	0.000002	0.004436
5	0.000002	0.000002	0.004513
...	...	...	...
2499	0.000929	0.001004	0.006122
2500	0.000929	0.001005	0.006156
2501	0.000928	0.001006	0.006179
...	...	...	...
4999	0.001567	0.001770	0.005193
5000	0.001566	0.001771	0.005177
5001	0.001567	0.001771	0.005162
...	...	...	...
9999	0.003301	0.003721	0.003344
10000	0.003317	0.003728	0.003340
10001	0.003316	0.003727	0.003340
...	...	...	...
19999	0.003861	0.004016	0.000002
20000	0.003737	0.003905	0.000002
20001	0.003621	0.003811	0.000002

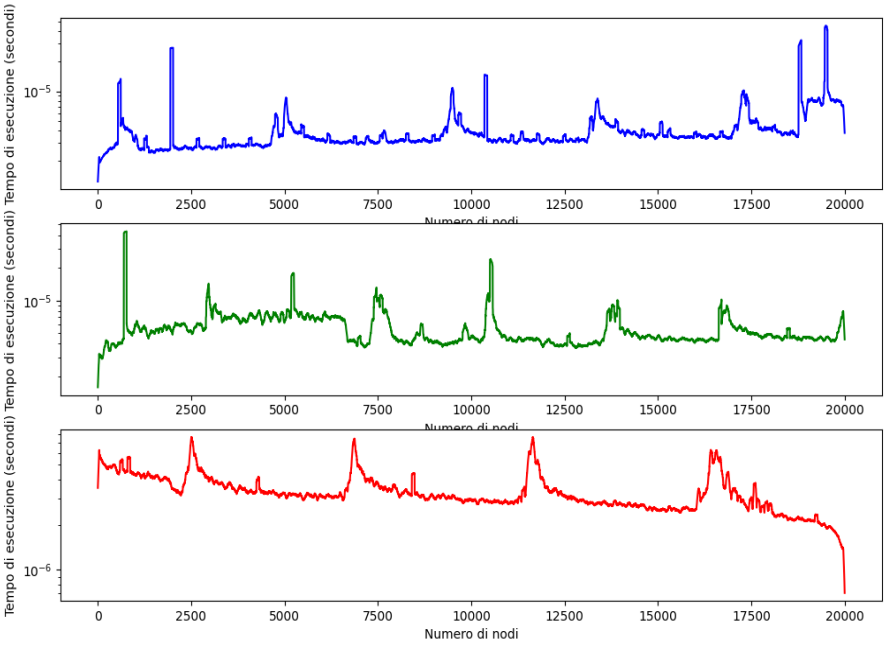

Figura 9, Grafico tempi di esecuzione di ogni nodo AC

Nel AC con una stampa in scala logaritmica possiamo notare che il grafico del tempo si comporta come una funzione logaritmica, quindi $O(log(n))$.

Tabella 3: Tempi di esecuzione nel caso peggiore (WC)

Numero Nodi	Aggiunta (s)	Recupero (s)	Eliminazione (s)
1	0.000001	0.000002	3.506668e-06
2	0.000001	0.000002	3.565334e-06
3	0.000001	0.000002	3.620001e-06
4	0.000001	0.000002	3.698668e-06
5	0.000001	0.000002	3.801334e-06
...	...	...	...
2499	0.000003	0.000005	7.417335e-06
2500	0.000003	0.000005	7.462669e-06
2501	0.000003	0.000005	7.504002e-06
...	...	...	...
4999	0.000006	0.000006	3.222666e-06
5000	0.000006	0.000006	3.230666e-06
5001	0.000007	0.000006	3.246666e-06
...	...	...	...
9999	0.000004	0.000005	2.938667e-06
10000	0.000004	0.000005	2.933334e-06
10001	0.000004	0.000005	2.930667e-06
...	...	...	...
19999	0.000004	0.000005	7.373320e-07
20000	0.000004	0.000004	7.213318e-07
20001	0.000004	0.000004	7.039992e-07

5.3 Hash

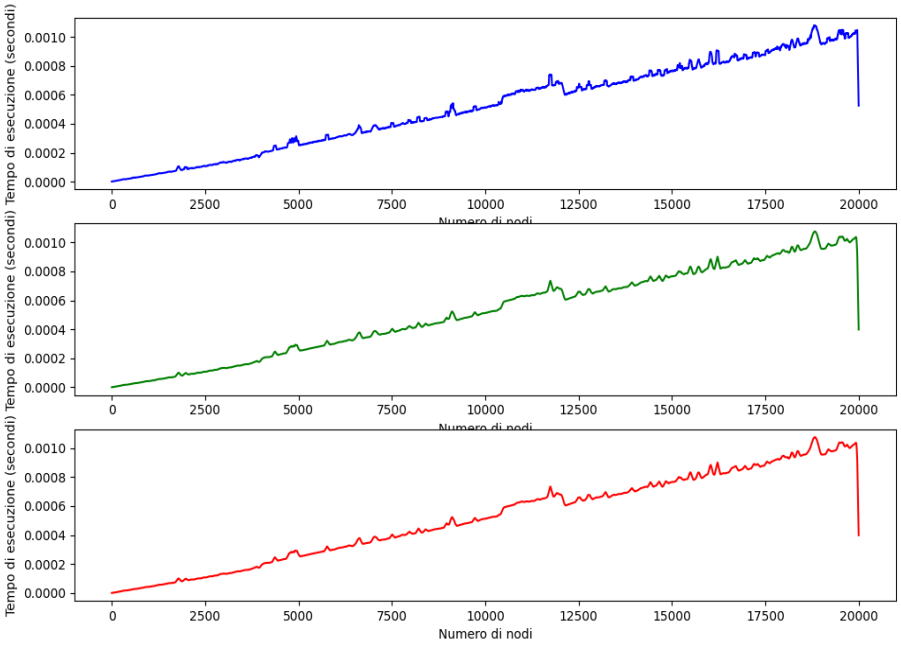

Figura 10, Grafico tempi di esecuzione di ogni nodo WC

Come detto in precedenza si comporta come una normale lista concatenata $O(n)$:

Tabella 4: Tempi di esecuzione nel caso peggiore (WC)

Numero Nodi	Aggiunta (s)	Recupero (s)	Eliminazione (s)
1	8.679996e-07	7.886044e-07	7.886044e-07
2	8.973332e-07	8.199821e-07	8.199821e-07
3	9.279993e-07	8.514844e-07	8.514844e-07
4	9.586662e-07	8.833244e-07	8.833244e-07
5	9.906664e-07	9.156266e-07	9.156266e-07
...	...	...	...
2499	1.060480e-04	1.075774e-04	1.075774e-04
2500	1.060973e-04	1.075694e-04	1.075694e-04
2501	1.061400e-04	1.075615e-04	1.075615e-04
...	...	...	...
4999	2.723533e-04	2.672248e-04	2.672248e-04
5000	2.707587e-04	2.668176e-04	2.668176e-04
5001	2.675613e-04	2.664261e-04	2.664261e-04
...	...	...	...
9999	5.131360e-04	5.128272e-04	5.128272e-04
10000	5.130413e-04	5.128489e-04	5.128489e-04
10001	5.128707e-04	5.128726e-04	5.128726e-04
...	...	...	...
19999	5.528133e-04	4.269142e-04	4.269142e-04
20000	5.392293e-04	4.129462e-04	4.129462e-04
20001	5.257427e-04	3.989721e-04	3.989721e-04

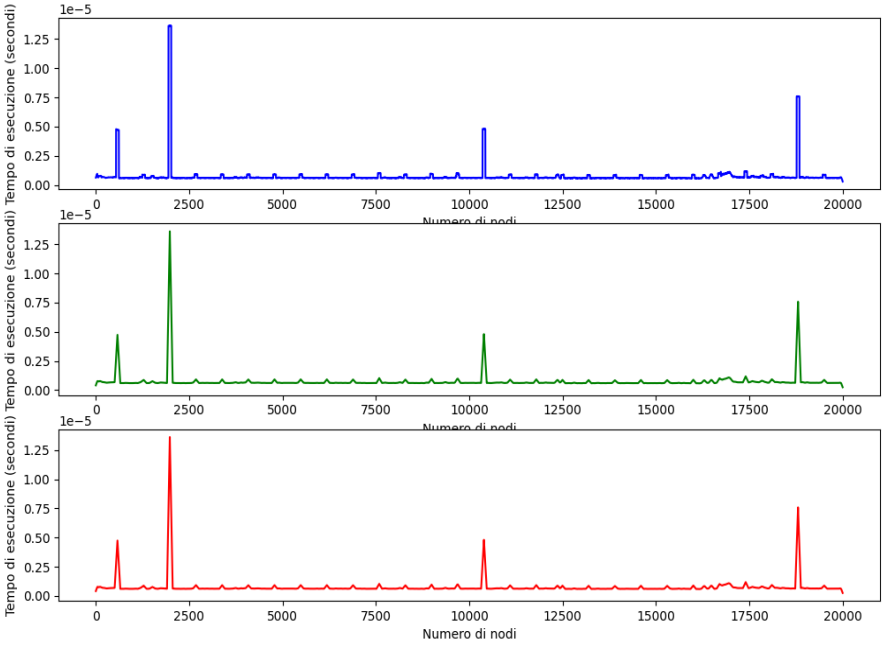

Figura 11, Grafico tempi di esecuzione di ogni nodo AC

Anche la tabella hash si comporta come ci si era ipotizzato, anche sono presenti delle curve anomale ma nonostante ciò ha un andatura approssimabile ad $O(1)$.

Tabella 5: Tempi di esecuzione nel caso peggiore (AC)

Numero Nodi	Aggiunta (s)	Recupero (s)	Eliminazione (s)
1	6.466662e-07	4.112176e-07	4.112176e-07
2	6.546662e-07	4.217954e-07	4.217954e-07
3	6.639996e-07	4.317865e-07	4.317865e-07
4	6.719997e-07	4.415821e-07	4.415821e-07
5	6.813331e-07	4.512532e-07	4.512532e-07
...	...	...	...
2499	6.186675e-07	6.069696e-07	6.069696e-07
2500	6.173342e-07	6.068985e-07	6.068985e-07
2501	6.173349e-07	6.068096e-07	6.068096e-07
...	...	...	...
4999	6.040007e-07	6.116277e-07	6.116277e-07
5000	6.040015e-07	6.117521e-07	6.117521e-07
5001	6.053348e-07	6.118410e-07	6.118410e-07
...	...	...	...
9999	6.226668e-07	6.200530e-07	6.200530e-07
10000	6.240001e-07	6.202841e-07	6.202841e-07
10001	6.226660e-07	6.200530e-07	6.200530e-07
...	...	...	...
19999	3.279994e-07	2.667730e-07	2.667730e-07
20000	3.199993e-07	2.580797e-07	2.580797e-07
20001	3.106667e-07	2.493863e-07	2.493863e-07

5.4 Grafici composti

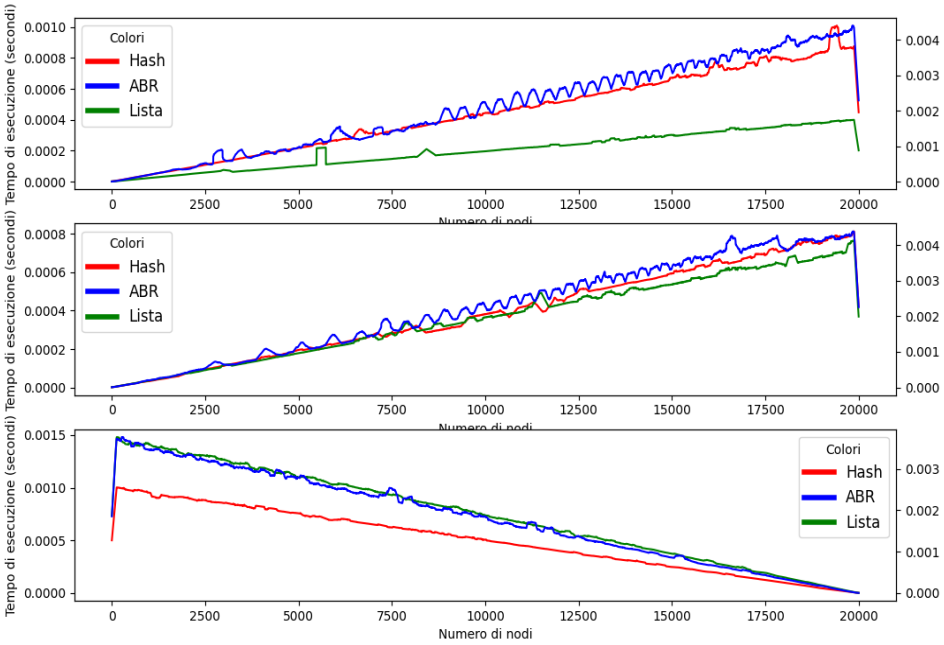

Figura 12, Grafico tempi di esecuzione di ogni nodo WC

L'andatura meno regolare dell'albero binario di ricerca è giustificata dal fatto che è stato scritto tramite delle funzione ricorsive, quindi può risultare più lento o avere un numero di picchi maggiore nel grafico rispetto alla lista concatenata ed hash.

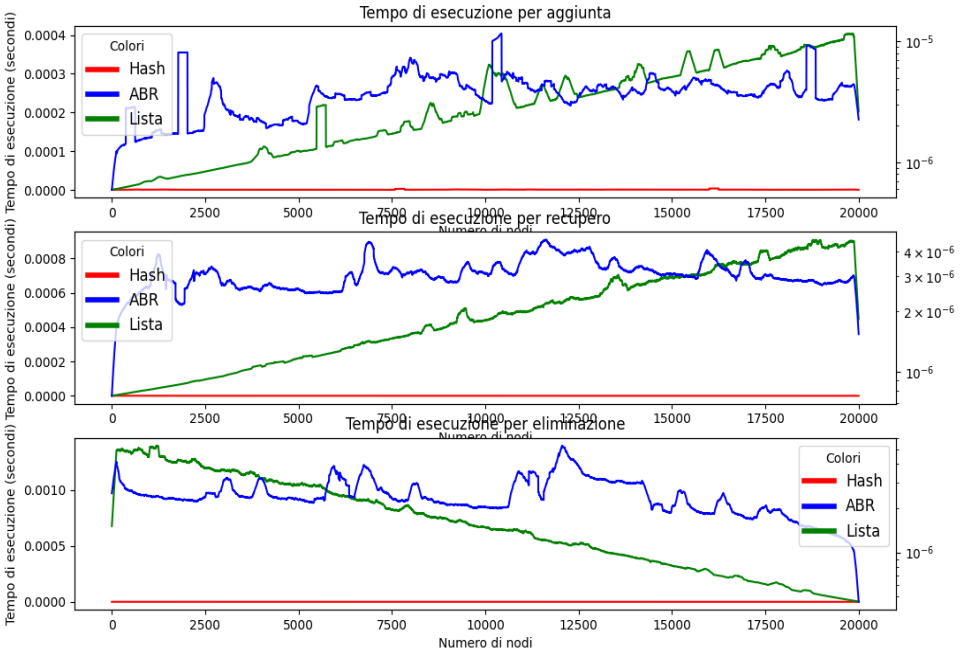

Figura 13, Grafico tempi di esecuzione di ogni nodo WC

In questo grafico possiamo notare con semplicità le varie andature nel caso medio di ogni struttura dati e confermare ulteriormento ciò che è stato descritto in precedenza, dato che l'ABR ha

un andatura logaritmica e meno regolare rispetto alla lista concantenata, la quale ha comunque più picchi rispetto all'hash.

Tabella 6: Tempi di esecuzione aggiunta (AC)

Numero Nodi	Albero Binario	Tabella Has	Lista Concatenta
1	0.000006	0.000003	0.000002
2	0.000006	0.000003	0.000002
3	0.000007	0.000003	0.000002
...	...	...	...
2499	0.001061	0.000325	0.000086
2500	0.001062	0.000325	0.000086
2501	0.001064	0.000325	0.000087
...	...	...	...
4999	0.002327	0.000434	0.000178
5000	0.002327	0.000434	0.000178
5001	0.002326	0.000433	0.000178
...	...	...	...
9999	0.004294	0.000849	0.000379
10000	0.004290	0.000848	0.000379
10001	0.004284	0.000848	0.000378
...	...	...	...
19999	0.004405	0.001382	0.000354
20000	0.004362	0.001371	0.000352
20001	0.004369	0.001359	0.000349

Tabella 7: Tempi di esecuzione recupero (AC)

Numero Nodi	Albero Binario	Tabella Hash	Lista Concatenta
1	0.000005	0.000004	0.000003
2	0.000005	0.000004	0.000003
3	0.000005	0.000004	0.000003
...	...	...	...
2499	0.001337	0.000225	0.000263
2500	0.001332	0.000225	0.000263
2501	0.001331	0.000226	0.000263
...	...	...	...
4999	0.002030	0.000481	0.000587
5000	0.002028	0.000481	0.000587
5001	0.002024	0.000482	0.000587
...	...	...	...
9999	0.004435	0.001046	0.001103
10000	0.004436	0.001047	0.001103
10001	0.004436	0.001048	0.001103
...	...	...	...
19999	0.004931	0.000992	0.001029
20000	0.004899	0.000984	0.001018
20001	0.004866	0.000976	0.001009

Tabella 8: Tempi di esecuzione elimina (AC)

Numero Nodi	Albero Binario	Tabella Hash	Lista Concatenta
1	0.003894	0.000971	0.001437
2	0.003931	0.000980	0.001448
3	0.003972	0.000989	0.001459
...	...	...	...
2499	0.007317	0.001285	0.002638
2500	0.007305	0.001284	0.002638
2501	0.007292	0.001284	0.002640
...	...	...	...
4999	0.006217	0.000973	0.002307
5000	0.006230	0.000973	0.002308
5001	0.006239	0.000974	0.002311
...	...	...	...
9999	0.003837	0.001053	0.001453
10000	0.003838	0.001052	0.001453
10001	0.003835	0.001052	0.001453
...	...	...	...
19999	0.000006	0.000003	0.000011
20000	0.000006	0.000003	0.000011
20001	0.000006	0.000003	0.000011

5.5 Conclusioni

Tutto ciò che è stato ipotizzato nella sezione Spiegazione toerica del problema è stato rispettato e provato nella sezione Descrizione dei risultati. Abbiamo provato che nel caso peggiore di ogni struttura dati testata si ha un andatura $O(n)$. Mentre nel caso medio dell'albero binario $O(log(n))$, nel caso medio della tabella hash $\Theta(1)$.

Riferimenti bibliografici

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein (2009) Introduzione agli algoritmi e strutture dati Terza edizione, McGraw Hill.