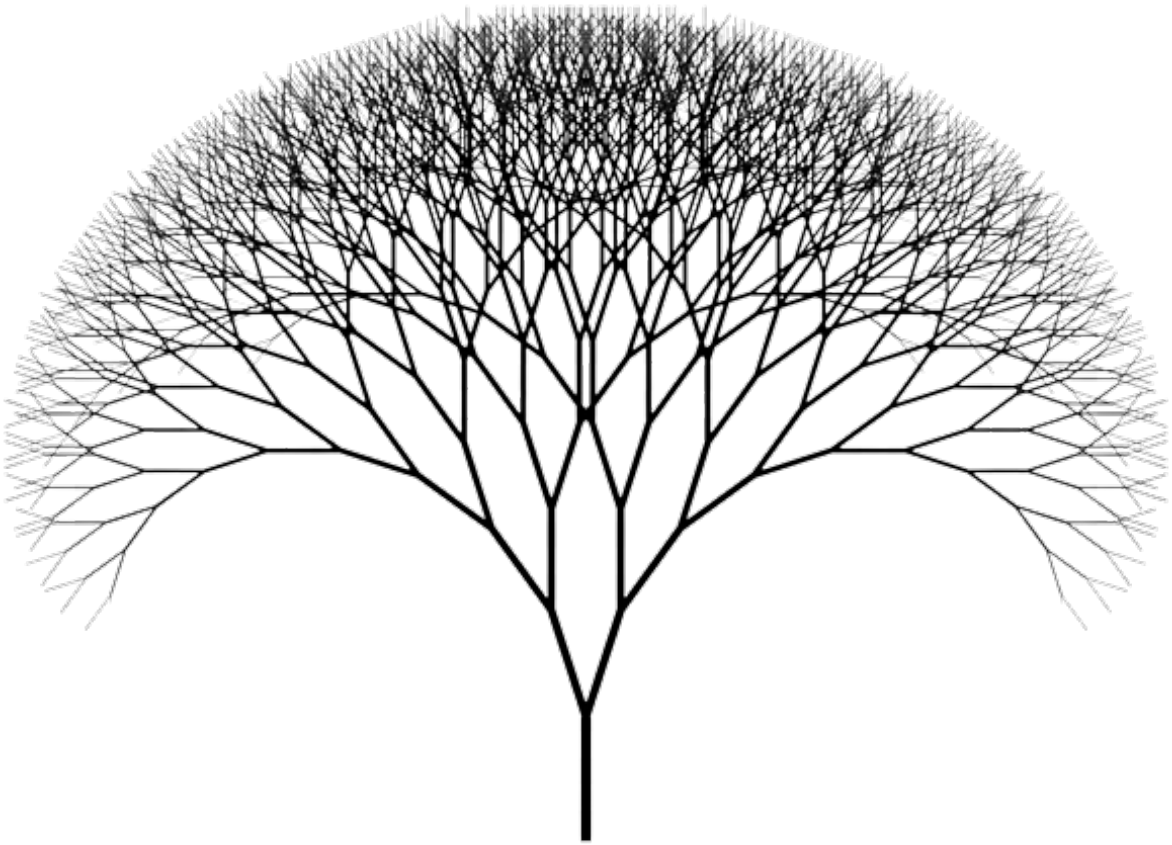
Alberi di Ricerca:
ABR, ARN e AVL

Laboratorio di Algoritmi a.a. 2022/2023

Leonardo Borsi



Indice

1 Introduzione

L'obiettivo del progetto consiste nel confrontare i vari metodi con cui può essere costruito un albero di ricerca, in particolare andremo ad analizzare le seguenti tipologie di alberi:

- Alberi Binari di Ricerca
- Alberi Rosso Neri
- Alberi AVL

Oltre che ad un'analisi delle classi python con cui vengono implementate le tipologie di albero, andremo a confrontare i costi e le prestazioni di ciascuno di essi con una serie di test riguardanti i due principali metodi che li caratterizzano: Inserimento e Ricerca.

2 Le strutture dati

2.1 Alberi Binari di Ricerca

Gli alberi binari di ricerca (ABR) sono strutture dati dinamiche costituite da un insieme di nodi, identificati da una chiave, ed un insieme di archi orientati, rappresentati come coppie ordinate di nodi; inoltre data una qualsiasi coppia di nodi, esiste un solo cammino che li unisce, in quanto gli archi dell'albero non formano mai cicli. Viene definito binario in quanto da ogni nodo possono uscire al massimo due archi, che identificano i legami col figlio destro e sinistro. Un albero binario di ricerca è tale se soddisfa le seguenti proprietà:

- Se il nodo A si trova nel sottoalbero di sinistra del nodo B, allora la chiave di A è minore della chiave di B
- Se il nodo A si trova nel sottoalbero di sinistra del nodo B, allora la chiave di A è maggiore o uguale della chiave di B

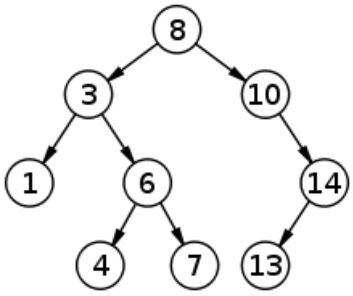


Figura 1: Albero binario di ricerca

2.2 Alberi Rosso Neri

Gli alberi rosso neri (ARN) sono degli alberi binari di ricerca i quali vanno ad implementare un'informazione aggiuntiva su ogni nodo, ovvero il colore: Rosso o Nero (dato rappresentabile con 1 bit). La differenza sostanziale con gli ABR sta nel metodo di inserimento, il quale prevede una funzione di bilanciamento dell'albero, a seguito dell'aggiunta di un nodo.
Questo bilanciamento serve per assicurare che l'ARN mantenga inviolate le seguenti proprietà:

- Ogni nodo deve essere rosso o nero
- La radice dell'albero è sempre nera
- Tutte le foglie T.Nill sono considerate come un nodo nero
- Tutti i cammini che portano dalla radice a una foglia hanno lo stesso numero di nodi neri
- Se un nodo è rosso, allora entrambi i figli sono neri

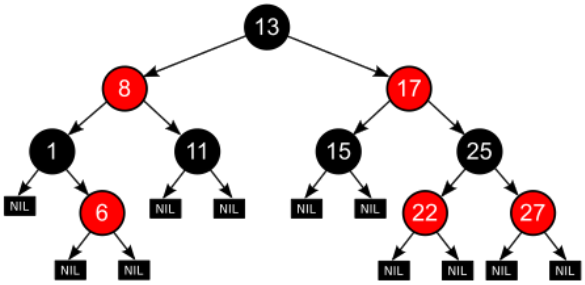


Figura 2: Albero rosso nero (Fonte: Wikipedia)

2.3 Alberi AVL

Gli alberi AVL sono degli alberi binari di ricerca bilanciati in altezza, overro degli alberi che rispettano la seguente proprietà:

- Per ogni nodo dell'albero, l'altezza del suo sottoalbero sinistro e del suo sottoalbero destro differiscono al più di uno

Quindi in un albero AVL, entrambi i sottoalberi destro e sinistro sono a loro volta alberi AVL. Come per gli alberi rosso neri, gli alberi AVL prevedono una funzione di bilanciamento a seguito dell'aggiunta di un nuovo nodo, in modo da mantenere inviolata la proprietà (e quindi mantenere l'albero bilanciato in altezza).

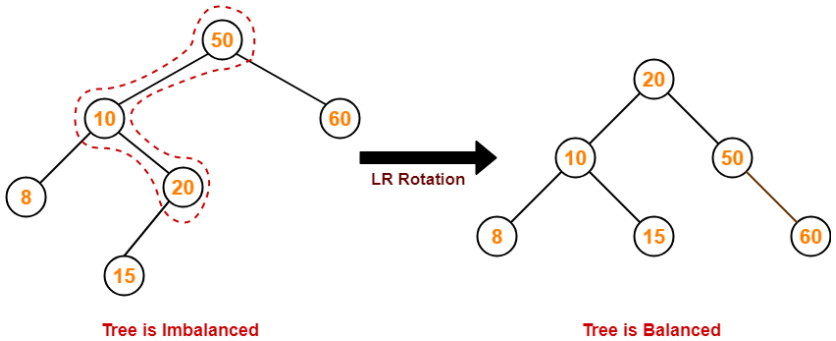


Figura 3: Bilanciamento in altezza di un albero per rispettare proprietà degli AVL (Fonte: GateVidyalay)

3 Le operazioni

3.1 Inserimento

Negli alberi binari di ricerca l'inserimento di un nuovo nodo avviene confrontando ricorsivamente la chiave del nodo da inserire con le chiavi dei nodi dell'albero: partendo dalla radice, ci sposteremo a destra o a sinistra a seconda del valore incontrato; una volta arrivati ad una foglia, dipendentemente dal suo valore, inseriremo il nuovo nodo come suo figlio destro o sinistro.
Come descritto in precedenza, negli ARN e AVL viene effettuata un'operazione di bilanciamento a seguito di questo inserimento, attuo a fare rispettare le regole della struttura dati.

3.2 Ricerca

La ricerca segue lo stesso iter dell'inserimento: data una chiave da cercare, essa viene confrontata ricorsivamente con i nodi dell'albero, partendo dalla radice e spostandosi sul figlio destro o sinistro a seconda che la chiave del nodo sia minore o maggiore della chiave cercata.
Se si arriva ad una foglia senza aver trovato la chiave di interesse, la ricerca ha esito negativo in quanto essa non è presente nell'albero. In questo caso l'operazione di ricerca è la medesima per tutte e tre le tipologie di albero.

3.3 Complessità delle operazioni

Le operazioni descritte hanno un costo in funzione dell'altezza dell'albero, e quindi richiedono un tempo di esecuzione proprozionale ad essa. L'altezza di un albero non è altro che il cammino radice-foglia più lungo tra tutti i cammini presenti, e proprio per questo costituisce la complessità delle operazioni nel caso peggiore. Negli ABR questa dipende soprattutto dall'ordine con cui vengono inseriti i nodi nell'albero:

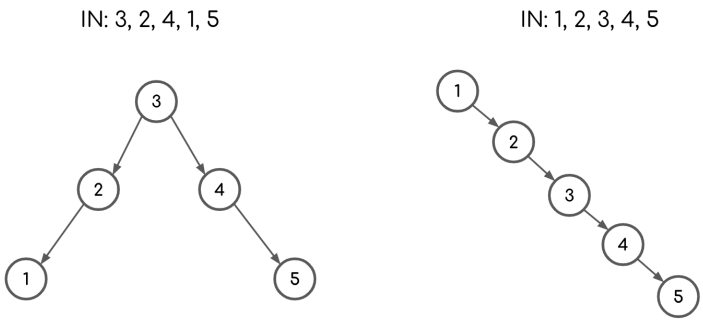


Figura 4: Altezza ABR a seconda dell'ordine di inserimento

Come possiamo vedere, avendo le stesse chiavi in ingresso ma inserite in un ordine diverso, si ottengono due alberi diversi: il primo bilanciato con un'altezza di 3, mentre il secondo sbilanciato con un'altezza di 5. Questo ci permette di individuare il caso peggiore per le operazioni che può capitare in un ABR, ovvero l'inserimento di una catena lineare di n chiavi, la quale genererà un albero di altezza pari a n.

Negli ARN e negli AVL abbiamo un comportamento diverso nel caso peggiore; infatti grazie alla funzione di bilanciamento a seguito di ogni inserimento, avremo un'altezza h che equivale a

$$h = log(n+1) - 1 = log(n)$$

Assumiamo quindi lg n come altezza nei casi medi di tutti gli alberi e nei casi peggiori per gli ARN e AVL. Riassumendo in una tabella i costi delle operazioni di inserimento e ricerca, per il caso peggiore e medio di ciascuna tipologia di albero:

	Inserimento		**Ricerca**	
	Caso peggiore	Caso medio	Caso peggiore	Caso medio
ABR	*O(n)*	*O(lg n)*	*O(n)*	*O(lg n)*
ARN	*O(lg n)*	*O(lg n)*	*O(lg n)*	*O(lg n)*
AVL	*O(lg n)*	*O(lg n)*	*O(lg n)*	*O(lg n)*

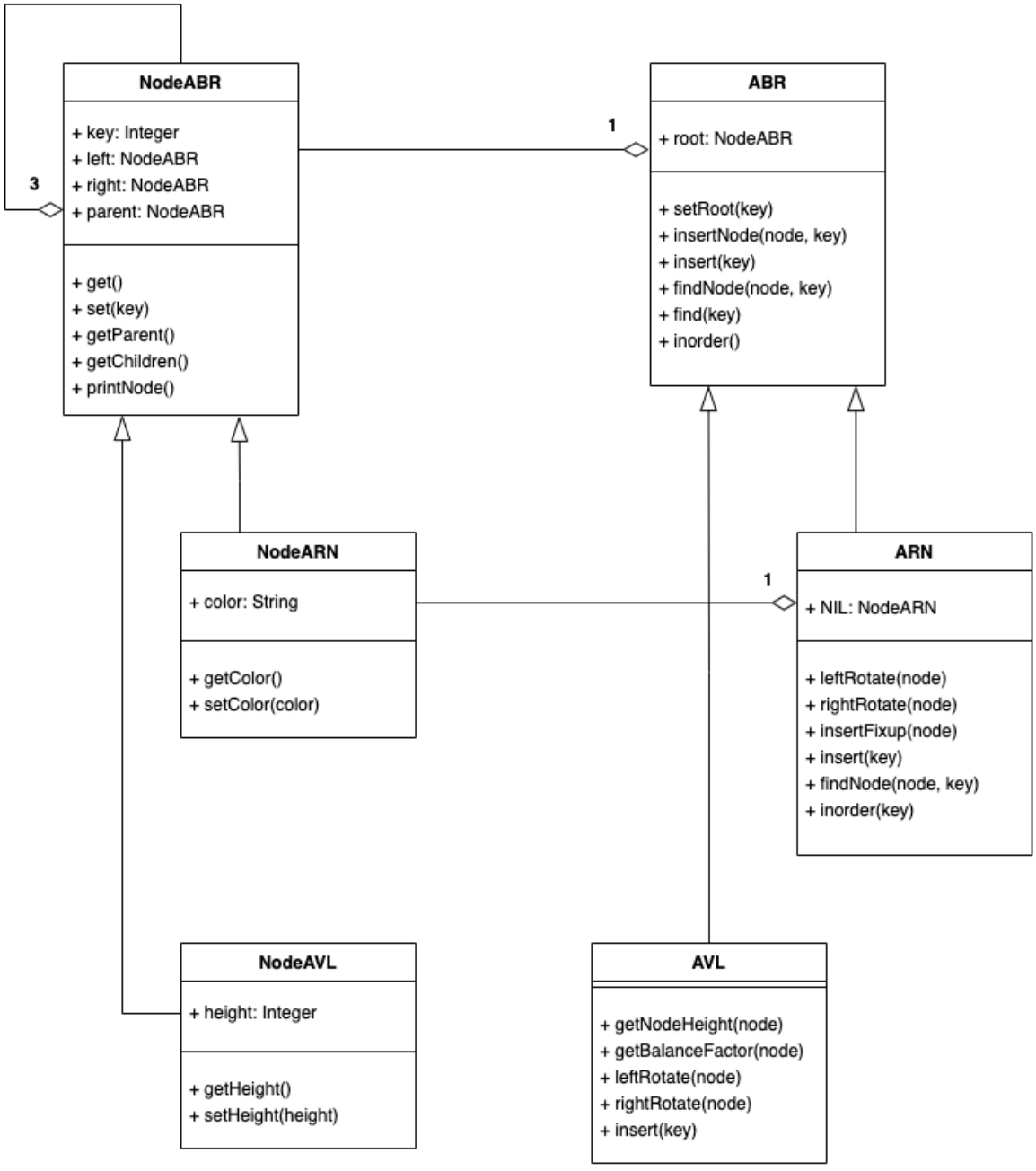
Tabella 1: Complessità in tempo delle operazioni su ciascun albero

Il nostro obiettivo quindi sarà verificare sperimentalmente la veridicità delle complessità descritte nella tabella, confrontando le prestazioni di ciascuna tipologia di albero in base al tempo reale che impiegano a completare le operazioni.

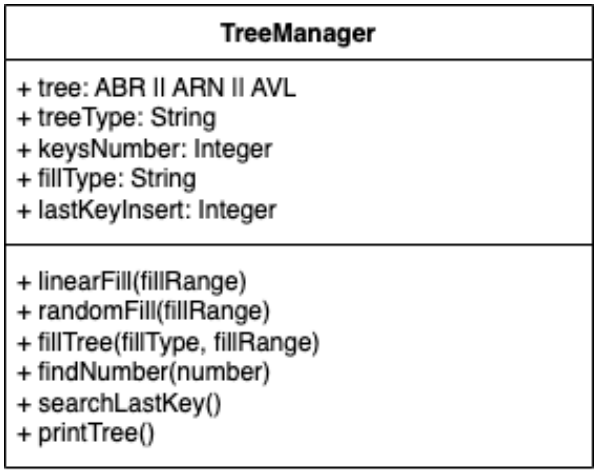
4 Implementazione delle strutture e dei test

4.1 Documentazione del codice

I tre alberi binari sono stati implementati tramite le seguenti classi:



Successivamente ho creato la classe TreeManager per poter gestire tutti i dati e le operazioni necessarie ad effettuare i test su un albero da una singola classe, indipendentemente dalla tipologia di esso.



Ogni istanza di TreeManager gestirà un singolo albero di una specifica tipologia e grandezza, e ci permetterà di testare le operazioni di inserimento e ricerca su di esso.

4.2 Descrizione degli esperimenti

Per sperimentare le operazioni ho suddiviso i test a seconda del numero di chiavi inserite nell'albero ed al tipo di inserimento.

Verranno creati alberi con:

- 50 elementi
- 250 elementi
- 500 elementi
- 900 elementi

e per ciascuna quantità di elementi, considereremo sia il caso con un inserimento lineare che quello con un inserimento randomico.
Il numero degli elementi è stato scelto in modo da poter rilevare una differenza nelle prestazioni, e cercando di non superare i 1000 elementi per evitare di avere più di un tot di ricorsioni (altrimenti il compilatore ritorna il seguente errore: RecursionError: maximum recursion depth exceeded while getting the str of an object).

Quindi a seconda del tipo di inserimento delle n chiavi:

- Caso Lineare: vengono inserite linearmente le chiavi da 0 ad n
- Caso Randomico: per poter confrontare coerentemente gli alberi e quindi inserire lo stesso set di chiavi nello stesso ordine in ciascun albero, ho creato una funzione statica per generare file contenenti set di chiavi randomiche che verranno usate per riempire gli alberi.
  Dato il range del valore delle chiavi da generare e la quantità, crea il file di test nella cartella inputs.

,

```
def createTestFile(poolRange, num):
    testFile = open('inputs/' + str(num) + '.txt', 'w')
    for x in range(0, num):
        testFile.write(str(random.randint(0, poolRange)) + '\n')
    print('Created test file ' + str(num) + '.txt')


def generateTests():
    createTestFile(50, 50)
    createTestFile(250, 250)
    createTestFile(500, 500)
    createTestFile(900, 900)
```

Figura 5: Funzione per generare i file contenenti i set di chiavi randomiche

Quindi richiamando la funzione statica generateTests() ci prepariamo i set di chiavi randomiche da utilizzare negli esperimenti, utilizzando le quantità indicate precedentemente ed un range identico ad essa (date n chiavi il range sarà 0-n), come avviene per l'inserimento lineare.

I dati descritti fin'ora, oltre che ad illustrarci come verrà testato l'inserimento, verranno usati come base per verificare la prestazione della ricerca. Infatti, dopo aver inserito tutte le chiavi negli alberi, effettuerò una ricerca dell'ultima chiave inserita su ciascuno di essi, in modo da cercare ogni volta una foglia. Avendo costruito gli alberi con i medesimi set di chiavi, inserite nello stesso ordine, potremo effettuare un benchmark preciso dei tempi di ricerca per ciascuna tipologia di albero.

4.3 Flusso e Misurazioni

Analizziamo il flusso del main del progetto:

- Inizialmente dichiariamo le quantità di elementi su cui effettueremo i test e le tipologie di inserimento/generazione delle chiavi

```
20  def main():
21
22      keysNumbers = [50, 250, 500, 900]
23      fillTypes = ['linear', 'random']
```

Figura 6: Definizione quantità elementi e tipologie inserimento

- Per ogni quantità ed ogni tipologia, inizializzeremo un albero di ogni tipo (ABR, ARN e AVL) e vi inseriremo le chiavi secondo i parametri attuali; il metodo fillTree restituisce gli array contenenti le chiavi inserite e i tempi di inserimento, i quali verrano utilizzati per popolare i grafici.

```
32      for n in keysNumbers:
33          for fillType in fillTypes:
34              ABR = TreeManager('ABR')
35              ARN = TreeManager('ARN')
36              AVL = TreeManager('AVL')
37              xABR, yABR = ABR.fillTree(fillType, n)
38              xARN, yARN = ARN.fillTree(fillType, n)
39              xAVL, yAVL = AVL.fillTree(fillType, n)
40              fig = plt.figure()
41              plt.xlabel('Chiavi inserite')
42              plt.plot(xABR, yABR)
43              plt.plot(xARN, yARN)
44              plt.plot(xAVL, yAVL)
45              plt.legend(['ABR', 'ARN', 'AVL'])
46              title = fillType + ' - insert - ' + str(n)
47              fig.savefig(title, dpi=300)
```

Figura 7: Inizializzazione degli alberi, riempimento e generazione grafici dell'inserimento

- I tempi per l'inserimento vengono calcolati dentro i metodi di riempimento dell'albero linearFill e randomFill, secondo la formula :

$$timeArray[i] = t_i + timeArray[i-1]$$

dove t_i è il tempo impiegato per inserire il nodo i

```
def linearFill(self, fillRange):
    totalTime = 0
    operationArray = [0]
    timeArray = [0]
    for v in range(0, fillRange):
        start = time.time()          Misurazione prima di inserire il nodo
        self.tree.insert(v)
        end = time.time()            Misurazione dopo aver inserito il nodo
        actualTime = end - start
        timeArray.append(actualTime + timeArray[v])
        operationArray.append(v)     Aggiungo il tempo effettivo per
        totalTime += actualTime      l'inserimento all'array delle
        self.lastKeyInsert = v       misurazioni
    return operationArray, timeArray
```

```
def randomFill(self, fillRange):
    numbers = getRandomNumbers(fillRange)
    operationArray = [0]
    timeArray = [0]
    totalTime = 0
    for i in range(0, fillRange):
        start = time.time()          Misurazione prima di inserire il nodo
        self.tree.insert(numbers[i])
        end = time.time()            Misurazione dopo aver inserito il nodo
        actualTime = end - start
        timeArray.append(actualTime + timeArray[i])
        operationArray.append(i)     Aggiungo il tempo effettivo per
        totalTime += actualTime      l'inserimento all'array delle
        self.lastKeyInsert = numbers[i]  misurazioni
    return operationArray, timeArray
```

Figura 8: Misurazione dell'inserimento di un nodo

- Successivamente andiamo a salvarci il tempo impiegato da ogni tipologia di albero per cercare l'ultima chiave inserita

```
48          if fillType == 'linear':
49              linearABRSearchTimes.append(ABR.searchLastKey())
50              linearARNSearchTimes.append(ARN.searchLastKey())
51              linearAVLSearchTimes.append(AVL.searchLastKey())
52          else:
53              randomABRSearchTimes.append(ABR.searchLastKey())
54              randomARNSearchTimes.append(ARN.searchLastKey())
55              randomAVLSearchTimes.append(AVL.searchLastKey())
```

Figura 9: Ricerca dell'ultima chiave inserita in ogni albero

- I tempi per la ricerca vengono calcolati nel metodo atto alla ricerca dell'ultima chiave inserita searchLastKey, il quale restituisce il tempo effettivo impiegato per trovare la chiave

```
def searchLastKey(self):
    start = time.time()          Misurazione prima di iniziare la ricerca
    self.findNumber(self.lastKeyInsert)
    end = time.time()            Misurazione dopo aver trovato la chiave
    actualTime = end - start
    return actualTime            Restituisco il tempo impiegato per trovare la chiave
```

Figura 10: Misurazione della ricerca di un nodo

- Infine andiamo a generare i grafici delle ricerche, utilizzando i tempi impiegati da ciascun albero a trovare la sua ultima chiave inserita, per ogni quantità e tipologia di inserimento

```
57      searchKeysNumbers = [0, 50, 250, 500, 900]
58      fig = plt.figure()
59      plt.xlabel('chiavi inserite')
60      plt.plot(searchKeysNumbers, linearABRSearchTimes)
61      plt.plot(searchKeysNumbers, linearARNSearchTimes)
62      plt.plot(searchKeysNumbers, linearAVLSearchTimes)
63      plt.legend(['ABR', 'ARN', 'AVL'])
64      title = 'linear - search'
65      fig.savefig(title, dpi=300)
66
67      fig = plt.figure()
68      plt.xlabel('chiavi inserite')
69      plt.plot(searchKeysNumbers, randomABRSearchTimes)
70      plt.plot(searchKeysNumbers, randomARNSearchTimes)
71      plt.plot(searchKeysNumbers, randomAVLSearchTimes)
72      plt.legend(['ABR', 'ARN', 'AVL'])
73      title = 'random - search'
74      fig.savefig(title, dpi=300)
```

Figura 11: Generazione grafici della ricerca

5 Analisi dei risultati

5.1 Inserimento di 50 chiavi

Analizzando i grafici ottenuti possiamo evincere che su un piccolo numeri di chiavi le differenze tra le prestazioni dei vari alberi siano ancora minime, però possiamo già notare 2 fattori:

- L'ABR, passate le 30 chiavi inserite, è l'albero che impiega più tempo per l'inserimento lineare
- L'AVL impiega più tempo dell'ARN, in particolare per l'inserimento randomico

Infatti nell'inserimento lineare, prima delle 30 chiavi inserite, l'ABR è più rapido dell'ARN e AVL nonostante si trovi nel caso peggiore; questo a causa del tempo richiesto dalle funzioni di bilanciamento degli altri due. Una volta che si passano le 30 chiavi, il caso peggiore inizia a "pesare" maggiormente, facendo aumentare il tempo richiesto per l'inserimento nell'ABR.

Nell'inserimento randomico l'ABR è il più efficente non dovendo bilanciare l'albero ad ogni chiave inserita, mentre l'AVL richiede più tempo dell'ARN a causa della sua funzione di bilanciamento più onerosa in termini temporali.

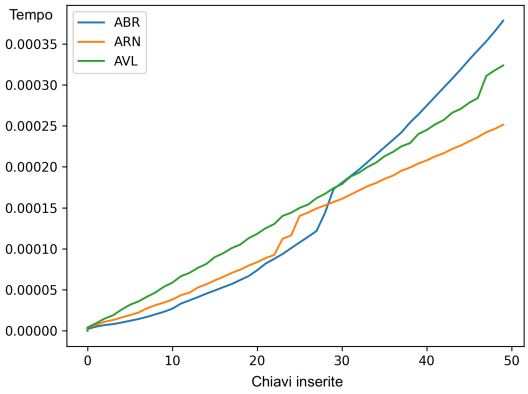


Figura 12: Inserimento Lineare di 50 chiavi

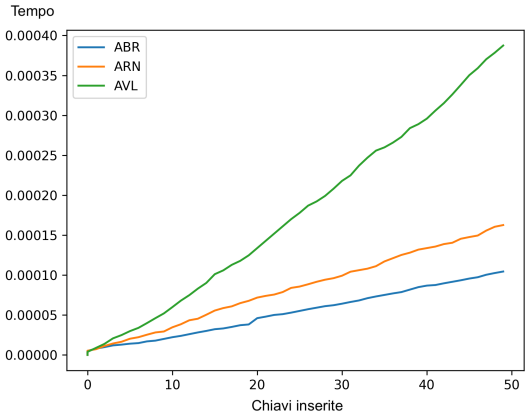


Figura 13: Inserimento Randomico di 50 chiavi

5.2 Inserimento di 250 chiavi

Passando alle 250 chiavi inserite possiamo notare come:

- La complessità dell'inserimento nel caso peggiore dell'ABR generi una curva esponenziale del tempo richiesto in funzione delle chiavi inserite
- In entrambe le tipologie di inserimento l'AVL continua a richiedere più tempo rispetto all'ARN

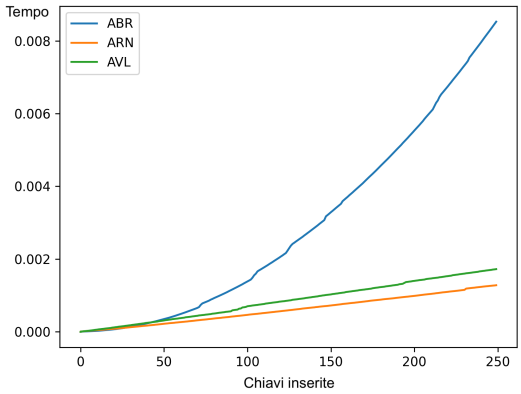


Figura 14: Inserimento Lineare di 250 chiavi

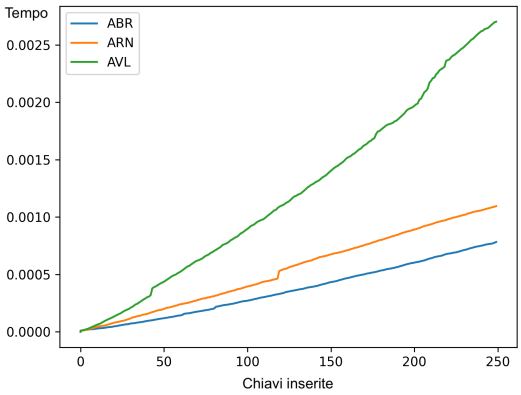


Figura 15: Inserimento Randomico di 250 chiavi

5.3 Inserimento di 500 chiavi

I grafici mantengono l'andatura del precedente caso; si accentua la curva esponenziale per l'inserimento lineare nell'ABR.

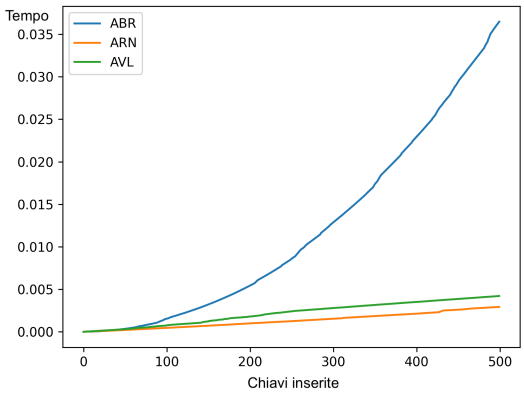


Figura 16: Inserimento Lineare di 500 chiavi

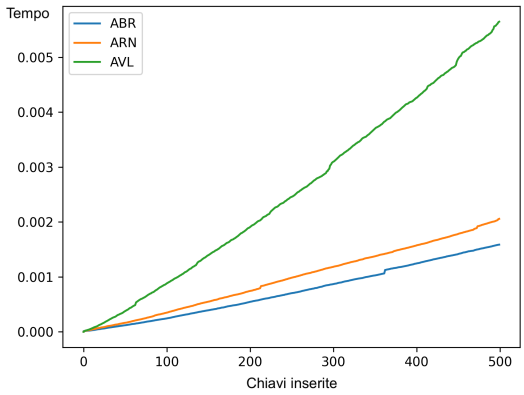


Figura 17: Inserimento Randomico di 500 chiavi

5.4 Inserimento di 900 chiavi

Con quest'ultimo test si riconfermano i grafici e le analisi descitte nei casi precedenti

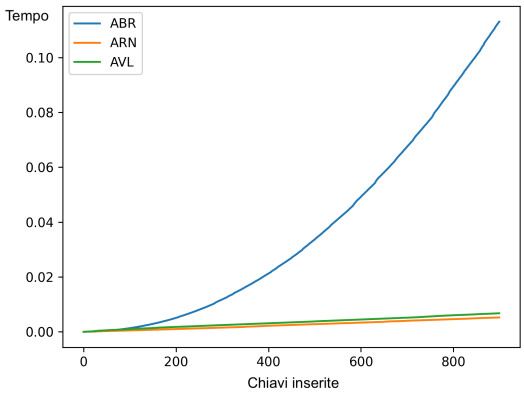


Figura 18: Inserimento Lineare di 900 chiavi

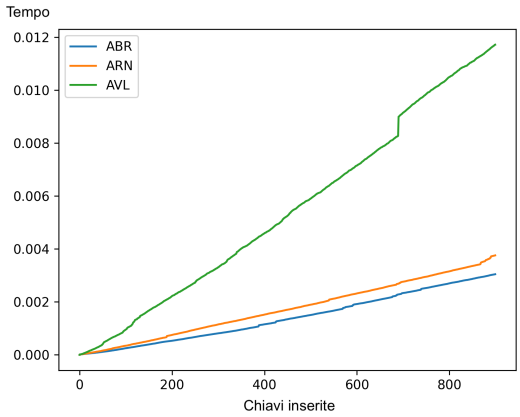


Figura 19: Inserimento Randomico di 900 chiavi

5.5 Ricerca

Per quanto riguarda il test effettuato per la ricerca di una chiave (in questo caso l'ultima chiave inserita per tutti i casi di inserimento visti prima), possiamo evincere che:

- Nel caso di inserimento lineare, l'ABR impiega un tempo notevolmente superiore in quanto, trovandosi nel suo caso peggiore, deve scorrere tutte le chiave inserite precedentemente per arrivare all'ultima inserita. L'AVL e l'ARN invece richiedono un tempo molto basso grazie al bilanciamento della loro struttura, in particolare l'AVL, che ha prestazioni leggermente superiori dovute al bilanciamento in altezza che effettua.

- Nel caso di inserimento randomico invece, a discapito del grafico ottenuto, rileviamo differenze nel tempo di ricerca veramente poco significative (nell'ordine di $1 * 10^{-}5$ secondi); ritroviamo comunque l'ABR come albero meno prestante nella ricerca.

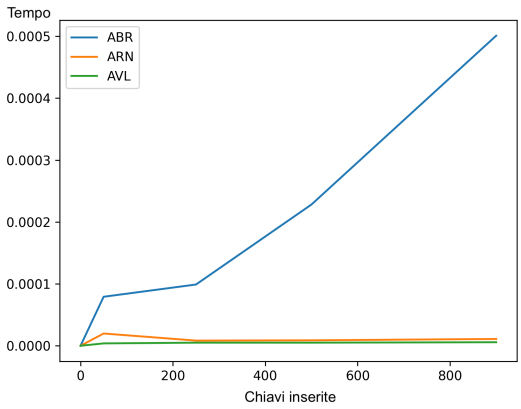


Figura 20: Ricerca dell'ultima chiave dopo l'inserimento lineare

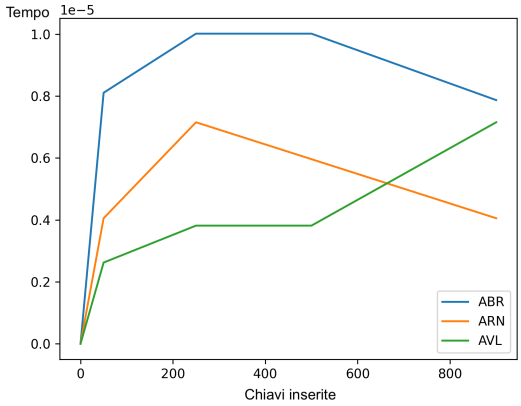


Figura 21: Ricerca dell'ultima chiave dopo l'inserimento randomico

6 Conclusioni

Alla luce degli esperimenti effettuati possiamo concludere che:

- l'ABR sia l'albero più prestante in termini di velocità quando ci troviamo nel caso medio per l'inserimento, grazie alla semplicità della sua funzione di insert, la quale però è anche causa della crescita esponenziale del tempo impiegato nel caso peggiore, nel quale invece risulta estremamente inefficiente rispetto agli altri alberi. Risulta inefficiente anche nel caso peggiore della ricerca, per cui impiega molto tempo a causa del suo sbilanciamento.
- l'ARN risulta l'albero più efficiente in linea generale per l'inserimento, in quanto è il più prestante quando ci troviamo nel caso peggiore, e nel caso medio si avvicina molto alle prestazioni dell'ABR, nonostante la funzione di bilanciamento che effettua all'aggiunta di ogni nuovo nodo. Grazie ad essa ha ottime prestazioni anche nella ricerca, in particolare nel caso peggiore.
- l'AVL invece si avvicina molto alle prestazioni dell'ARN nel caso peggiore dell'inserimento, ma allunga di molto i tempi quando ci troviamo nel caso medio, a causa della sua funzione di bilanciamento la quale richiede più tempo. Questa caratteristica torna però utile quando ci troviamo ad effettuare una ricerca, infatti, grazie ad un miglior bilanciamento in altezza, l'AVL risulta il più prestante nel caso peggiore.