Relazione Progetto in Itinere I

Corso di Ingegneria Degli Algoritmi

Matteo Esposito
Matricola: 0238638

Dicembre 2017

Indice

Abstract 1

1.1 Premessa

L'algoritmo presentato vorrebbe tentare di risolvere il problema N.2 il quale chiedeva:

"Siano dati due alberi AVL A e B tali che le chiavi di uno siano tutte strettamente minori delle chiavi dell'altro. Progettare e implementare un algoritmo che restituisca un nuovo albero AVL ottenuto come concatenazione di A e B."

L'approccio risolutivo consta dell'utilizzo di operazioni pre-esistenti, forniteci attraverso le classi e gli snipset presenti sulla pagina di GitHub del Corso, con una classe "extendedAVL" che riporta alcune funzioni il cui scopo é estendere ulteriormente la classe AVLDict. Il tempo di esecuzione totale nel caso peggiore risulta essere $O(log(n))$ in quanto ogni singola operazione svolta nel caso peggiore risulta essere eseguita per un tempo pari all'$O(log(n))$.

Il codice é cosí strutturato:

- main.py: contenente essenzialmente le DEMO dell'algoritmo
- la classe "ProjUtilities":contenente le principali classi ed il core dell'algoritmo stesso (la funzione *concatenate*)
- la succitata classe extendedAVL
- tutte le librerie e classi fornite precedentemente attraverso la pagina di GitHub.

L'Algoritmo 2

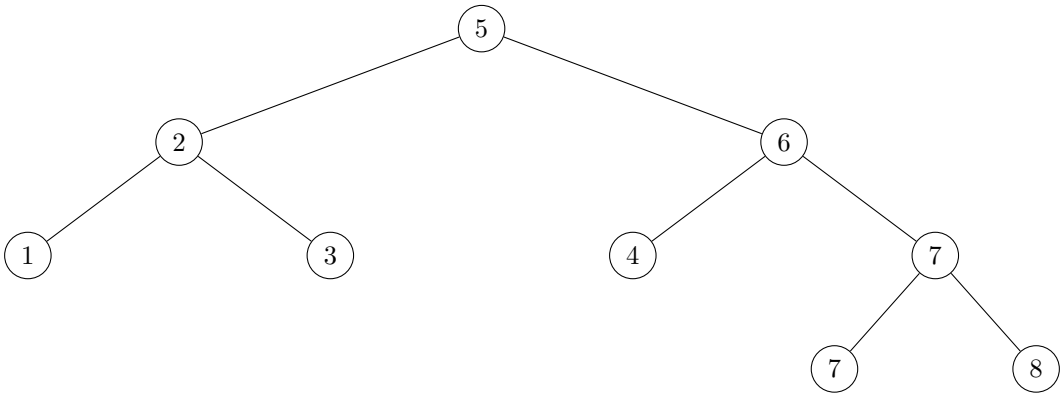
Attenzione: per brevità, verrá trattato estensivamente nella descrizione solo il caso in cui B sia piú alto di A, il caso inverso, comunque implementato nel codice, é simmetrico al presente e verrà presentato all'interno di parentesi quadre, in forma breve, per completezza. Il Tempo indicato come sotto titolo ad ogni sezione è il risultato dell'analisi teorica del codice.

2.1 Fase Preliminare (SOLO VERSIONE DEMO)

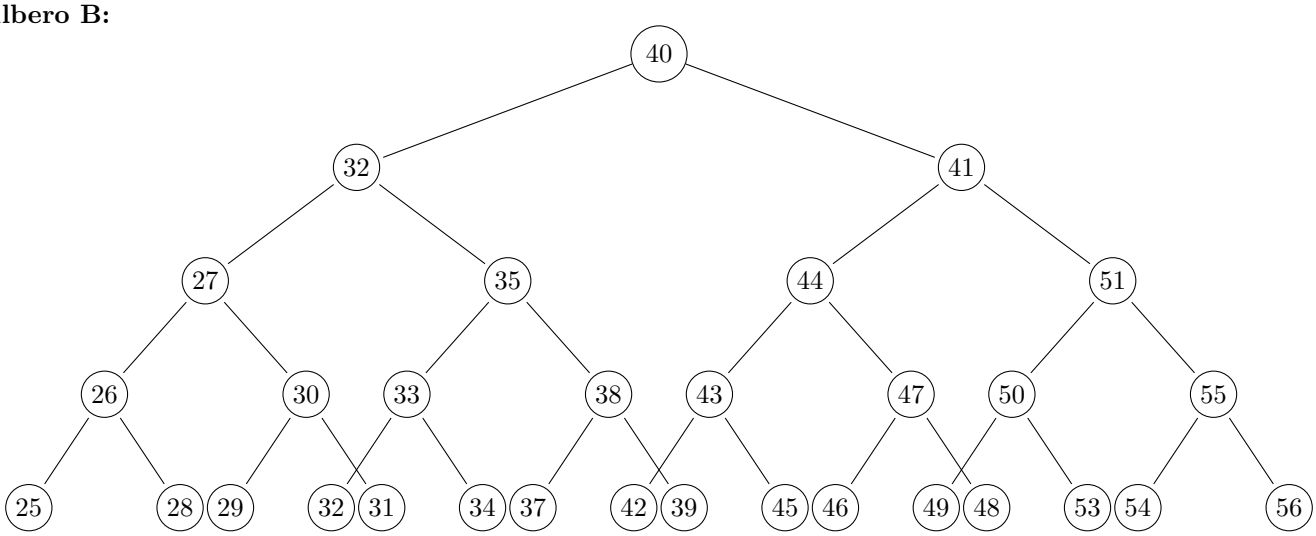
Tempo : $\mathbf{O(n)}$

Nella fase preliminare l'algoritmo crea o da un Array attraverso il metodo *createAVLTreeByArray()* o in maniera pseudo random 2 alberi AVL attraverso la funzione *generateRandomAVLTree* (contenuta in ProjUtilities). Il presente metodo, per quanto veloce possa essere é l'unico il cui tempo di esecuzione sia pari ad un O(n). Chiaramente é un metodo esistente solo nella DEMO dell'algoritmo mentre la sua implementazione pura non prevede affatto questa parte ragion per cui nel calcolo totale ho ignorato tale risultato.

Albero A:



Albero B:



2.2 Confronto

Tempo : **O**(**c**)

L'algoritmo, attraverso il metodo "getTreeHeight()" ottiene l'altezza dei due alberi AVL e ne salva il contenuto in due variabili H_a ed H_b. (Tale richiesta ovviamente impiega un Tempo costante quindi assimilabile ad un O(1)). Con queste variabili salvate effettua un piccolo e breve confronto per stabilire quali dei due alberi sia il piú alto il risultato di tale confronto è alla base di tutto l'algoritmo. Per ipotesi, dal confronto risulta che A possegga un'altezza pari a 3 mentre l'albero B di 4.

2.3 Ricerca del massimo di A

Tempo : **O**(**log**(**n**))

L'idea alla base dell'algoritmo é quella di cercare di eseguire meno operazioni possibili, quindi, rispetto ad una prima idea naive che consisteva semplicemente nell'inserire i singoli nodi da un albero all'altro, si cerca invece di innestare per intero l'albero piú basso, per ipotesi A, in quello piú alto. Per far ció é necessario trovare un nodo che ci consenta di fare tale operazione senza sbilanciare troppo l'albero altrimenti il vantaggio dell'innesto dell'albero andrebbe perso nelle tante operazioni di rotazioni e di inserimento. L'attuale *aim* quindi sarebbe cercare un nodo al quale si possa innestare l'intero albero A. Quel nodo fa effettivamente già parte di A in quanto se scegliessimo il Massimo di A l'intero albero potrebbe esservi innestato come figlio sinistro. Ora peró va cercato nell'albero B un nodo R dove innestare il nuovo albero così creato. Dall'esempio sopra riportato il massimo di A, conservato nel suo nodo piú a destra, é pari ad 8.
[Se A fosse più alto di B allora si dovrebbe cercare invece il minimo di B, infatti si può innestare l'intero albero B alla destra del suo stesso minimo. Quindi il Nodo R non apparterrà a B in questa ipotesi bensì ad A]

2.4 Ricerca del Nodo R

Tempo : **O**(**log**(**n**))

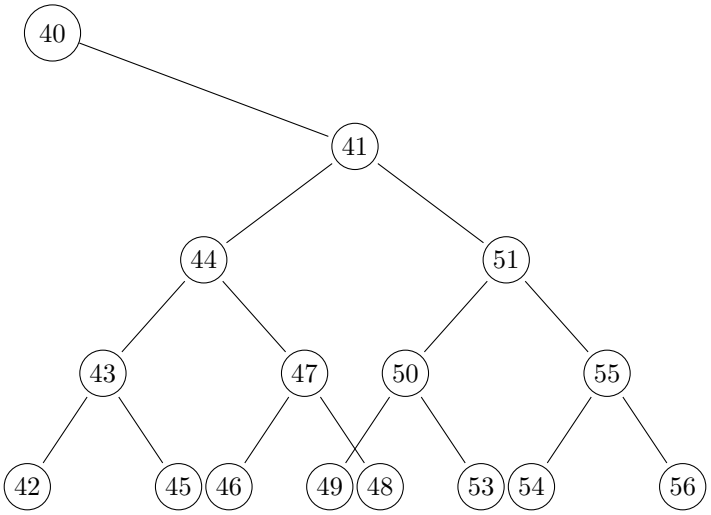
Il prossimo *aim* consiste nel trovare il nodo R di B dove innestare l'albero A cosí ri-arrangiato e fare tutto ció senza sbilanciare troppo l'albero B. Una soluzione é riutilizzare l'informazione sull'altezza dell'albero A poiché sicuramente se riuscissimo ad innestare l'albero ad un nodo di altezza eguali o, nella peggiore delle ipotesi, maggiore di una singola unità, ció ci consentirebbe di ottenere un albero B non troppo sbilanciato. Nell'ipotesi di B più alto di A ovviamente i candidati sarebbero 2 un nodo appartenente alla parte sinistra dell'albero ed uno alla destra. Poiché il nostro obbiettivo é innestare un albero con chiavi tutte minori di B sicuramente il candidato ideale é proprio il nodo della parte sinistra dell'albero. Tale nodo sarà per noi il nodo R. Nell'esempio il nodo che corrisponde a detti criteri é propri il figlio sinistro della radice (32).
[Nell'ipotesi A più alto di B il candidato si trova nella parte destra dell'albero di A. Sfruttando il fatto che B ha tutte chiavi maggiori di A è ottimale trovare un nodo nella parte destra di A che di fatti sarà sicuramente maggiore di A e minore di B]

2.5 Eliminazione del sotto-albero del nodo R

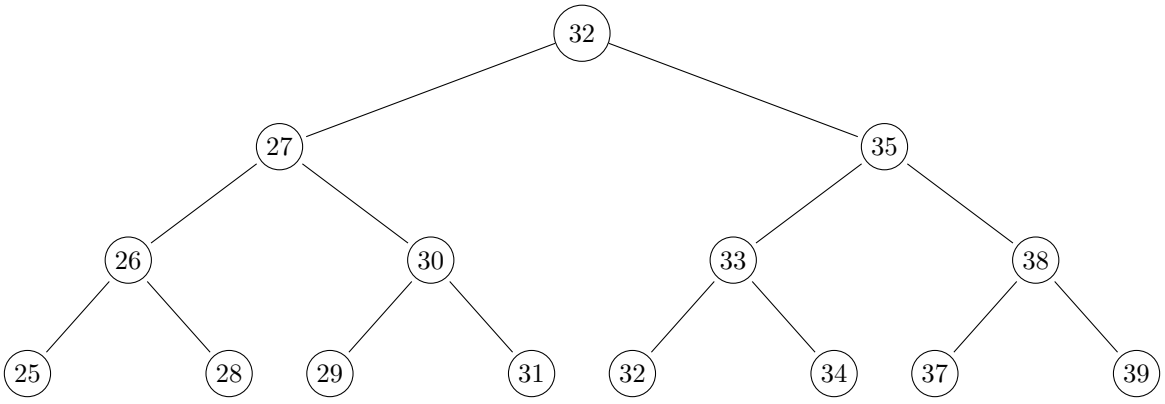
Tempo : **O**(**c**)

Nella peggiore delle ipotesi il nodo R ha un figlio sulla sua destra e sulla sua sinistra quindi sarebbe necessario trovare un modo con cui innestare l'intero albero come figlio del nodo R senza però violare la proprietà degli Alberi Binari di Ricerca per cui ogni nodo non puó avere pió di due figli. Una buona strategia quindi consisterebbe nell'eliminazione del sotto albero di R (con R incluso) (operazione che impiega $O(log(n))$) sostituire quindi a quel nodo il Massimo di A, con l'intero albero di A come suo figlio sinistro. Dall'esempio segue l'albero B senza il nodo R ed il sotto-albero di R: [Scegliendo come nodo R il candidato alla destra di A il nuovo albero da innestare in A avrà come radice il minimo di B, alla sua destra l'intero albero B e alla sua sinistra appunto il nodo R con suo relativo sotto-albero]

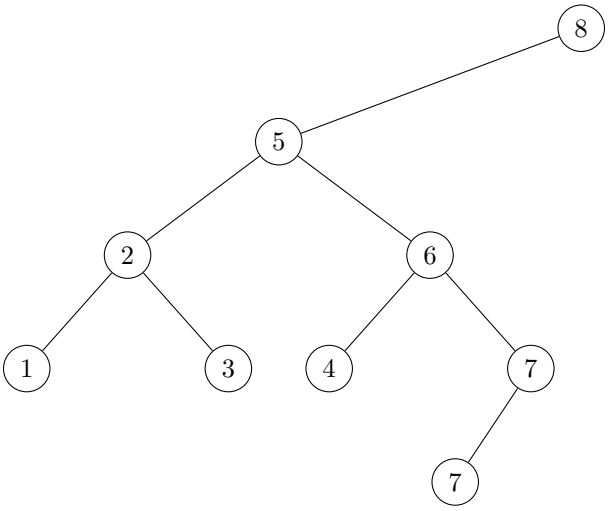
Albero B privato del Nodo R con relativo sotto-albero:



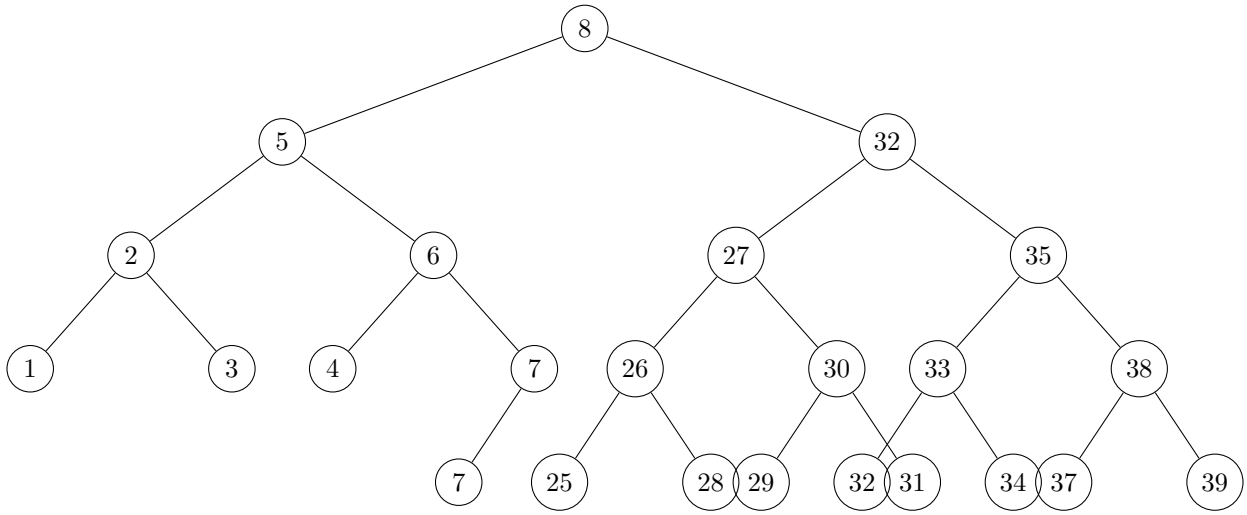
Nodo R e suo Sotto-albero:



Albero A Riarrangiato:



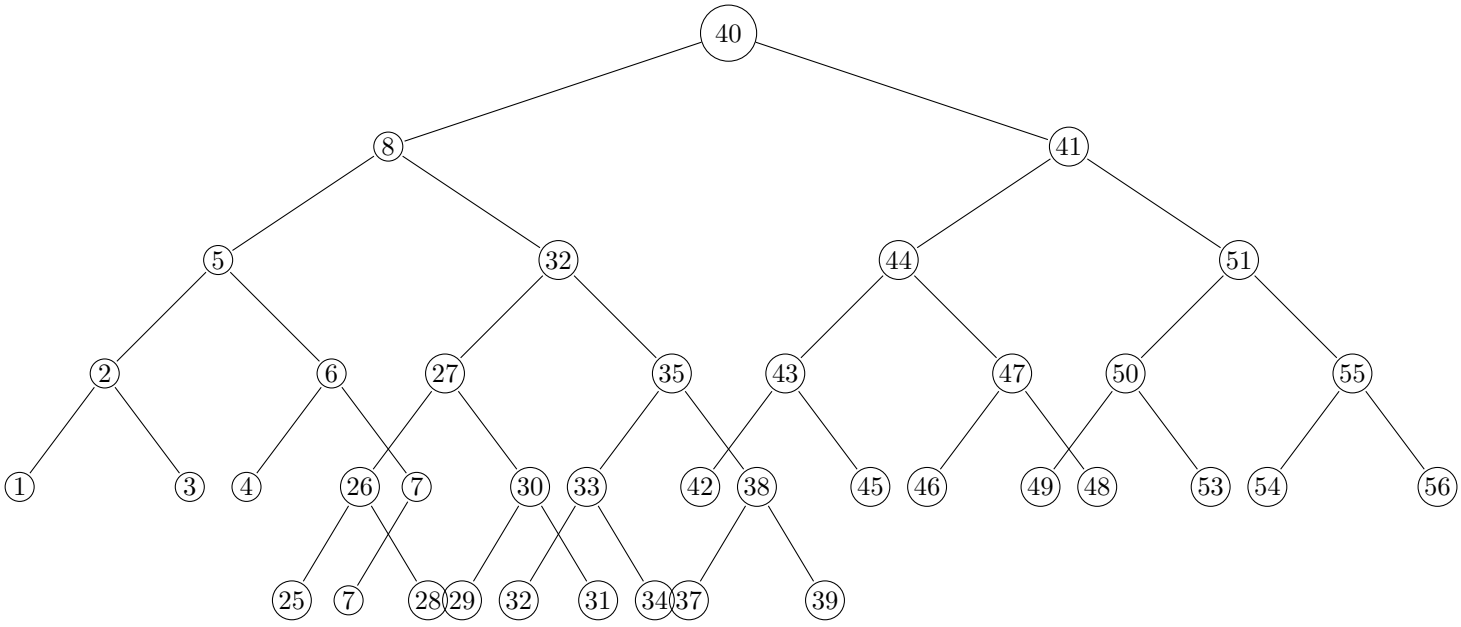
Nuovo Albero da Innestare in B:



2.6 Innesto del Sotto-albero R

Tempo : **O(c)**
Ora il il nuovo nodo così innestato rimane sprovvisto di figlio destro. Sfruttando la proprietá che ogni chiave dell'albero B é maggiore delle chiavi dell'albero di A possiamo innestare come figlio destro il nodo R con il suo intero sotto albero. [L'albero così creato in precedenza può essere inserito come figlio destro del padre di R senza perciò violare alcuna proprietà degli Alberi AVL salvo l'altezza che verrà subito aggiornata]

Alberi Concatenati:



2.7 Aggiornamento dell'altezza

Tempo : **O(log(n))**

Per costruzione il nodo così creato (figlio sx albero A e figlio dx nodo R con relativo sotto-albero) ha altezza eguali al vecchio nodo R, nella peggiore delle ipotesi differenti per una unità, si puó chiedere quindi attraverso il metodo *balInsert()* di ri-bilanciare l'albero così correggendo eventuali altezze discordi e così con l'ultima operazione in tempo $O(log(n))$ si ottengono i due alberi concatenati. Nell'esempio, il nuovo sotto-albero con radice in 8 é alto una singola unità in più del relativo sotto-albero radicato in 41 siccome è concesso uno sbilanciamento di una singola unità l'albero risulta difatti anche bilanciato di suo, completando il ciclo di operazioni in tempi logaritmici

Analisi delle Prestazioni: Caso Peggiore 3

3.1 Analisi Teorica

L'analisi delle prestazioni, nel caso peggiore, consente di fatto di stabilireé quanto sia piú o meno performante un dato algoritmo. L'analisi di questo algoritmo in particolare consta di diversi snipset di codice che contengono prevalentemente operazioni il cui tempo di esecuzione sia pari all'$O(log(n))$.

Codice

```
1         H_a = A.getTreeHeight()
2         H_b = B.getTreeHeight()
3         if H_a <= H_b :
4                 # Codice Caso H_a <= H_b
5         else:
6                 # Codice Caso H_a > H_b
```

La funzione Concatenate chiama un'altra funzione che ritorna l'altezza della radice Ottenute le due altezze fa un brevissimo confronto per stabilire in quale caso ricade l'attuale esecuzione quindi procede con il blocco di codice adatto.

Codice

```
1         def getTreeHeight(self):
2             return self.height(self.tree.root)
```

La funzione getTreeHeight, sovrabbodnate rispetto alla funzione height, restituisce in Tempo Costante O(1) l'altezza dell'albero ritornando essenzialmente l'altezza della radice

Attenzione: per coerenza e continuità con quanto esposto prima nella descrizione dell'algoritmo continueremo con l'ipotesi che l'albero B sia più alto dell'albero A

Codice

```
1         R_a = A.getMaxNode()
2         R_av = A.value(R_a)
3         R_ak = A.key(R_a)
4         A.delete(A.key(R_a))
```

Ora viene estratto il massimo da A
Ne viene salvato il valore
ed il valore key
e viene elimianto da A

Codice

```
1     def getMaxNode(self):
2          curr = self.tree.root
3          while curr.rightSon != None:
4              curr = curr.rightSon
5          return curr
6
```

La funzione getMaxNode percorre la parte destra dell'albero fino a restituire il nodo più a destra Nella Peggiore delle Ipotesi, questo metodo consta di un'esecuzione temporale pari all'$O(log(n))$, data che la peggiroe delle ipotesi è che debba scendere per l'intero albero il quale, per via delle prorpietà degli AVL ha un'altezza pari a log(n)

Codice

```
1     R_b = B.searchLeftForNodeOfHeight(H_a)
2      R_bt = B.tree.cut(R_b)
```

Viene quindi ricercato nella parte sinistra dell'albero B un nodo che abbia altezza eguali o di una singola unità più alto, nell'ipotesi peggiore, quindi si stacca dall'albero con il suo intero relativo sotto albero

Codice

```
1      def searchLeftForNodeOfHeight(self, height):
2          curr = self.tree.root
3          while self.height(curr) != height
4          and self.height(curr) != height+1:
5              if curr.leftSon  == None:
6                  break
7              else:
8                  curr = curr.leftSon
9          if self.height(curr.leftSon) == height:
10             return curr.leftSon
11         else:
12             return curr
```

La funzione searchLeftForNodeOfHeight ricerca a sinistra di un dato albero un nodo che possegga altezza eguali o di una singola unità in più rispetto allìaltezza chiesta. Nella peggiore delle ipotesi l'algoritmo scenderà fino ad una foglia, in tal caso, il tempo di esecuzione di tale istanza sarà pari all'altezza dell'albero stesso che, grazie alle proprietà degli Alberi AVL, sará pari al log(n)

Codice

```
1   tempA = createAVLByArray([R_av])
2   tempA.tree.insertAsLeftSubTree(tempA.tree.root,A.tree)
3   tempA.tree.insertAsRightSubTree(tempA.tree.root, R_bt)
4   tempA.updateHeight(tempA.tree.root)
```

Ora l'agoritmo crea un nuovo albero temporaneo, la funzione createAVLByArray, nella peggiore delle ipotesi impiega un tempo lineare all'input, ma in questo particolare caso la peggiore delle ipotesi è irrealizzabile infatti creando un albero di un singolo elemento, impiegherà tempo costante. O(1)) l'inserimento come figlio sinistro e destro invece constano anch'essi di operazioni di tempo costante. Quindi ne aggiorna l'altezza, tale operazione può essere fatta in tempi costanti ma nella peggiore delle ipotesi può richidere tempo logaritmico

Codice

```
1   B.tree.insertAsLeftSubTree(FtR_b, tempA.tree)
2   B.balInsert(FtR_b.leftSon)
```

Viene quindi inserito come figlio sinistro al padre del nodo R l'intero albero temporaneo precedentemente creato quindi si chiede di ribilanciare verso l'alto tale operazione nella peggiore delle ipotesi impiegherà un tempo di esecuzione logaritmico

Riepilogo: Il codice così analizzato ha evidenziato il fatto che qualsiasi altra chiamata a funzioni nuove da me create per estender ancora di più la classe AVLDict, nello scenario della funzione principale *concatenate* risultano costare tutte tempo logaritmico. Quindi l'analisi teorica ci conferma che l'algoritmo asintoticamente, quindi per grandi quantitá di dati, è un $O(log(n))$

3.2 Analisi Reale

Per dare una "dimostrazione" di quanto detto prima, ovvero che il tempo dell'algoritmo è un O(log(n)) ho pensato di realizzare un test, chiedendo al time di python di misurare 1000 chiamate (con elementi decrescenti) ed osservarne il tempo di esecuzione.

3.3 Alcuni Dati

Nella tabella a fianco si possono osservare subito alcuni dati rilevanti scelti tra le prime e le ultime iterazioni. Ad ogni iterazione, per cercare di pesudo-randomizzare i dati alcune parti della funzione *generateRandomAVLTree* venivano modificate ad-hoc im tempo reale in relazione all'attuale i-esima iterazione. Tutti i dati raccolti durante le simulazioni sono poi stati caricati in un foglio excel pronti per essere elaborati da Matlab per ottenere un grafico che possa mettere a confronto numero di elementi/-tempo di esecuzione con il numero di elementi/$log_2(n)$. Notevole di fatti è il dato del tempo di esecuzione (espresso in secondi) tranne in rare occasioni, da 110.000 elementi a 110, si potrebbe dire che è rimasto "costante nel suo andamento logaritmico" rispetto alla quantità di elementi nell'input.

Tempo (secondi):	Elementi Totali
0.000335216522217	110000
0.00032901763916	109890
0.00035285949707	109780
0.000355005264282	109670
0.000331878662109	109560
0.000336170196533	109450
0.000330924987793	109340
0.000339031219482	109230
0.00033712387085	109120
.....	.....
0.000196933746338	1210
0.000233888626099	1100
0.000180959701538	990
0.000180006027222	880
0.000179052352905	770
0.000180006027222	660
0.000165939331055	550
0.000160932540894	440
0.000162124633789	330
0.000144958496094	220
0.000126838684082	110

Tabella 3.1: Dati presentati in ordine discendente. (Il primo rappresenta la 1000-esima iterazione mentre l'ultimo dato la prima)

3.4 Caratteristiche della Macchina:

I test sono stati effettuati su di un computer notte-tempo aò fine di evitare qualsiasi interferenza da altri processi per l'accesso al disco o alle altre risorse del computer. La macchina su cui ho effettuato i test presenta le seguenti caratteristiche:

- Produttore Apple - Modello: 27"Mid2010
- Processore: 2,8 GHz Intel Core i5
- Memoria: 32 GB DDR3 1333MHz - Disco: SSD Samsung 500 GB
- Sistema Operativo: OS X El Capitain (Versione: 10.11.6 (15G17023))

3.5 Andamenti a Confronto

Attraverso Matlab, ho ottenuto il seguente grafico che evidenzia ancora di più l'andamento temporale dell'algoritmo e del log_2(numero di elementi).

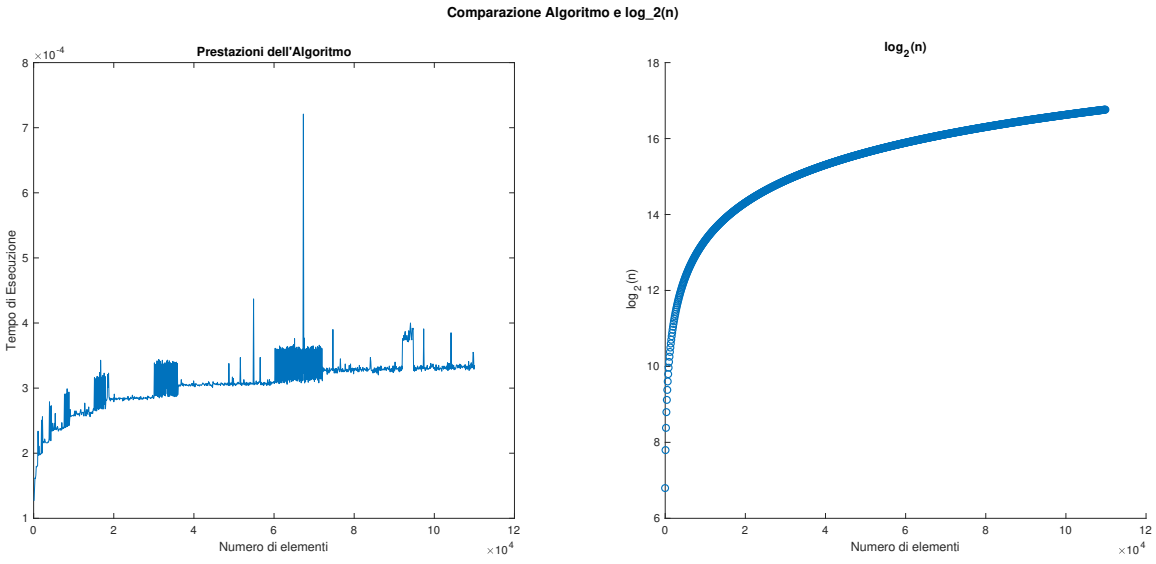


Figura 3.1: Grafico che mette a confronto numero di elementi/tempo di esecuzione e numero di elementi/log_2(#elementi)

Conclusione 4

4.1 Riflessioni Finali

L'interezza dell'algoritmo è stata così presentata in questa "breve" relazione. Al fine di aiutare nella comprensione del codice e della relazione stessa ho scritto effettivamente alcune funzioni sovrabbondati rispetto al codice fornitoci dai tutor, magari sprecato memoria nella creazione di un albero temporaneo od omesso alcuni passaggi di secondaria importanza dell'algoritmo in questa stessa relazione, tali scelte però sono state dettate da una criterio puramente "didattico" per aiutare nella migliore comprensione del codice. Tutto il codice è rilasciato sotto la MIT License.

Matteo Esposito