Esercizio A: Alberi rosso-neri vs Alberi Binari di Ricerca

Giovanni Stefanini - 6182949

Aprile 2021

1 Introduzione

In questo esercizio sono state testate le performance degli alberi binari di ricerca (ABR) e degli alberi rosso-neri (ARN), per avere dati sperimentali generati con Python, ai fini di valutare nella pratica i loro vantaggi e svantaggi.
A tal fine sono stai implementati prima le strutture dati che realizzando ABR e ARN che sono contenuti rispettivamente in *ABR.py* e *ARN.py*. Dopo è stata implementata una funzione, che permettesse di raccogliere tutti i dati, contenuta in *Test2ASD.py*. `graficiTuttiICasi()`, testa il caso peggiore di inserimento e ricerca per ABR, poi testa il caso medio di inserimento e ricerca per ABR, in più testa l'attraversamento, in seguito testa il caso peggiore di inserimento e ricerca per ARN, in fine testa il caso medio di inserimento e ricerca per ARN, in più testa l'attraversamento.

In particolare nei test sono stati testati alberi di dimensioni crescenti da 1 a 1000 aventi nodi con chiavi in ordine crescente per il caso peggiore, e nodi con chiavi in ordine casuale per il caso medio. Come elemento di riferimento per la qualità dell'algoritmo è stato preso il **tempo di esecuzione**.
Dentro la funzione di test, è stato messo come tempo limite di esecuzione 30 minuti.

2 Alberi binari di ricerca

Gli **alberi binari di ricerca**, o ABR, sono alberi che possiedono al massimo due figli per nodo ed hanno attributi `key`, `left`, `right` e `p`. La loro proprietà principale consiste nel fatto che, dato un nodo x, se y si trova nel sottoalbero sinistro di x, allora $y.key \leq x.key$, oppure, se y si trova nel sottoalbero destro di x, allora $y.key \geq x.key$. Detta h l'altezza dell'albero ed n il numero totale di nodi, le principali operazioni ABR hanno tempi:

Inserimento: $O(h)$
Ricerca: $O(h)$
Attraversamento inorder: $\Theta(n)$

Negli alberi completamente sbilanciati (in cui ogni nodo è figlio sinistro, oppure ogni nodo è figlio destro), generati inserendo nodi con chiavi in ordine non decrescente (solo figli destri) oppure non crescente (solo figli sinistri), l'altezza h dell'albero è pari al numero di nodi n, pertanto inserimento e ricerca richiedono, nella pratica, $\Theta(n)$. In ogni caso, l'altezza dipende fortemente dall'ordine con cui vengono inseriti i nodi; assumeremo che, inserendo nodi con chiavi dai valori casuali (senza quindi un ordinamento preciso), avremo un albero binario approssimativamente bilanciato.

3 Alberi rosso-neri

Gli **alberi rosso-neri**, o ARN, sono alberi binari di ricerca in cui ogni nodo possiede un attributo aggiuntivo, `color`: questo, implementato come un flag booleano, rappresenta appunto il "colore" del nodo (*rosso* se `True`, *nero* se `False`). Inoltre, ogni NIL è sostituito da un nodo, $T.nil$. Oltre alle proprietà tipiche di un ABR, esistono cinque proprietà RN:

1. Ogni nodo è rosso o nero
2. La radice è nera
3. Ogni foglia ($T.nil$) è nera
4. Se un nodo è rosso, allora entrambi i suoi figli sono neri
5. Tutti i cammini da ogni nodo alle foglie contengono lo stesso numero di nodi neri

Dato che un ARN è sempre quasi del tutto bilanciato (a differenza di un ABR) ed ha $h = lg(n)$, le sue operazioni hanno tempi:

Inserimento: $O(lg(n))$
Ricerca: $O(lg(n))$
Attraversamento inorder: $\Theta(n)$
Rotazioni: $O(1)$

Le rotazioni sono particolari operazioni locali costituite da sole modifiche a puntatori (da cui il tempo costante), che vengono utilizzate nella funzione ausiliaria di inserimento `InsertFixup` con lo scopo di bilanciare l'albero.

4 Specifiche tecniche

Per realizzare l'esercizio, i due tipi di albero sono stati implementati nel linguaggio Python nella versione presentata ai capitoli 12 e 13 del libro: Cormen, Leiserson, Rivest, Stein. *Introduzione agli Algoritmi e Strutture Dati, 3a edizione*. McGraw-Hill.
Le funzioni utilizzate per eseguire i test sugli alberi generano, in 2 cicli for, alberi di dimensioni sempre crescenti (fino ad un massimo di 1000 nodi) su cui vengono applicate le operazioni principali descritte nel paragrafo precedente, in più è stato utile aggiungere una operazione che restituisse l'altezza dell'albero. Il ciclo for interno è servito per ottenere una media di 10 esecuzioni, al fine di avere valori più attendibili.
Per ogni tipo di albero, vengono creati due grafi: uno i cui nodi vengono inseriti con chiavi in ordine crescente, per ottenere il caso peggiore ABR (perchè ARN in realtà resta bilanciato a prescindere dall'ordine con cui vengono inseriti i nodi), ed un altro in cui i nodi hanno chiavi con valore casuale e sono perciò meno condizionati dall'ordine di inserimento. Il caso migliore di ABR non è stato considerato in quanto consisterebbe in un albero completamente bilanciato, e sarebbe perciò equivalente ad un ARN completamente pieno (il quale a sua volta è semplicemente un normale ARN, ma pieno su tutti i livelli: ai fini pratici i tempi di inserimento, ricerca ed attraversamento non cambierebbero).
I tempi di esecuzione per ogni operazione, misurati senza includere azioni non attinenti come la generazione dei nodi da inserire o il salvataggio degli alberi, vengono salvati nelle rispettive variabili che vengono stampate alla fine di ogni ciclo for (prima che il numero di nodi aumenti).

Le **specifiche hardware e software** della macchina utilizzata per eseguire i test sono:

Scheda madre: X580VD Scheda di base ASUSTeK COMPUTER INC.
CPU: Intel Core i7-7700HQ CPU - 2.80GHz, 2808 Mhz, 4 core, 8 processori logici
RAM: 16 GB
SSD: SanDisck SD8SN8U128G1002 - 120 GB
HDD: TOSHIBA MQ04ABF100 - 1 TB
SO: Microsoft Windows 10 Home
IDE: JupyterLab 2.2.6

5 Simulazione e Risultati

5.1 Caso peggiore di inserimento: Si mostra di seguito il tempo di esecuzione nel caso peggiore, ottenuto inserendo nodi con chiavi in ordine crescente, dei due algoritmi su un albero con numero di nodi crescenti da 1 a 1000.

Tabella a confronto:

Numero di valori	Altezza ABR	Altezza ARN	Tempo peggiore inserimento ABR (ms)	Tempo peggiore inserimento ARN (ms)
0	-1	0	0.00051	0.00041
100	89	10	1.47122	0.88496
200	179	12	6.10156	1.93652
300	269	13	12.51018	3.13891
400	359	14	22.65388	4.1492
500	449	14	37.68413	5.72785
600	539	14	50.04458	6.52832
700	629	14	68.24103	7.99793
800	719	15	88.03676	9.09115
900	809	15	116.36335	10.43415
999	897	15	139.35333	11.60277

Figura 1: Tabella di ABR e ARN per Inserimento (caso peggiore)

Grafici a confronto:

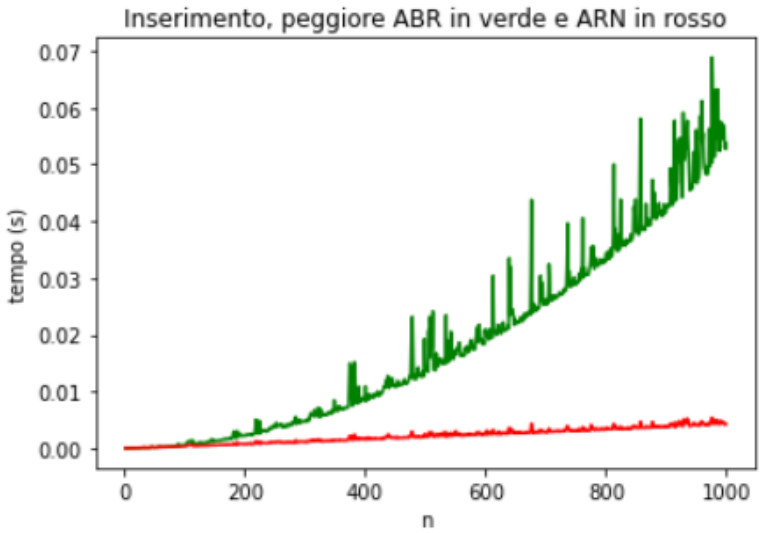

Figura 2: Grafici del tempo t (s) di ABR e ARN per Inserimento (caso peggiore)

Osservazioni:

Il caso peggiore dell'inserimento, ottenuto inserendo nodi con chiavi in ordine crescente, mostra come questa operazione sia drammaticamente lenta all'aumentare dei nodi in un albero binario di ricerca sbilanciato, soprattutto se confrontata con la sua performance su un albero rosso-nero: il tempo massimo per un ABR, calcolato con 1000 nodi, è stato di $139ms$, ovvero circa 10 volte maggiore rispetto ai $11ms$ impiegati dall'ARN per lo stesso numero di elementi.
Notiamo inoltre che l'altezza dell'albero ARN è molto inferiore (15) rispetto all'altezza dell'albero ABR (897) . Come previsto, quindi, l'inserimento rischia di diventare un'operazione costosa per gli ABR: qualora sia probabile che l'ordine dei nodi da inserire sia quasi esclusivamente non (de)crescente, la scelta del tipo di albero da utilizzare deve obbligatoriamente favorire gli alberi rosso-neri.

5.2 Caso peggiore di ricerca: Si mostra di seguito il tempo di esecuzione nel caso peggiore, ottenuto inserendo nodi con chiavi in ordine crescente, dei due algoritmi su un albero con numero di nodi crescenti da 1 a 1000.

Tabella a confronto:

Numero di valori	Altezza ABR	Altezza ARN	Tempo peggiore ricerca ABR (ms)	Tempo peggiore ricerca ARN (ms)
0	-1	0	0.00043	0.00047
100	89	10	0.03496	0.00586
200	179	12	0.0693	0.0067
300	269	13	0.10266	0.00711
400	359	14	0.14792	0.0077
500	449	14	0.18038	0.00778
600	539	14	0.13149	0.00749
700	629	14	0.18303	0.00646
800	719	15	0.2686	0.00858
900	809	15	0.38784	0.00923
999	897	15	0.37365	0.0096

Figura 3: Tabella di ABR e ARN per Ricerca (caso peggiore)

Grafici a confronto:

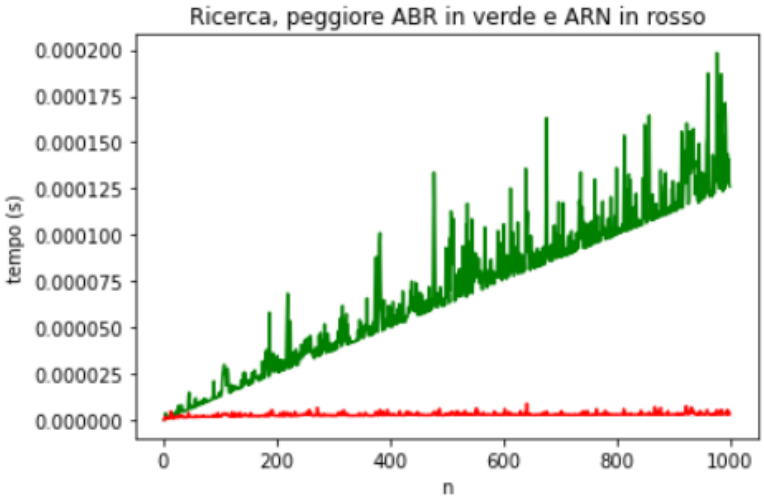

Figura 4: Grafici del tempo t (s) di ABR e ARN per Ricerca (caso peggiore)

Osservazioni:

La ricerca nel caso peggiore, eseguita per la chiave massima dell'albero (che si trova quindi nell'ultimo livello) costituisce il secondo caso in cui la scelta di un albero rosso-nero è palesemente migliore se i nodi vengono inseriti con un certo ordinamento. Pur essendo un'operazione veloce rispetto all'inserimento ($0.373ms$ vs. i $139ms$ per 1000 nodi dell'inserimento ABR), è comunque molto lenta se confrontata con la stessa operazione applicata su un ARN delle stesse dimensioni, che impiega $0,0096ms$.

5.3 Caso random di inserimento: Si mostra di seguito il tempo di esecuzione nel caso medio (ottenuto inserendo nodi in ordine casuale) dei due algoritmi su un albero

con numero di nodi crescenti da 1 a 1000.

Tabella a confronto:

Numero di valori	Altezza ABR	Altezza ARN	Tempo inserimento casuale ABR (ms)	Tempo inserimento casuale ARN (ms)
0	-1	0	0.00043	0.00051
100	11	7	0.28591	0.60011
200	14	8	0.67545	1.29992
300	16	9	1.07547	2.04217
400	15	10	1.54612	3.10036
500	17	10	2.10785	3.7612
600	16	10	2.33655	4.37636
700	17	11	2.76034	5.15997
800	18	11	3.30706	5.97065
900	18	11	3.86242	6.70494
999	19	11	4.36858	7.68206

Figura 5: Tabella di ABR e ARN per Inserimento (caso random)

Grafici a confronto:

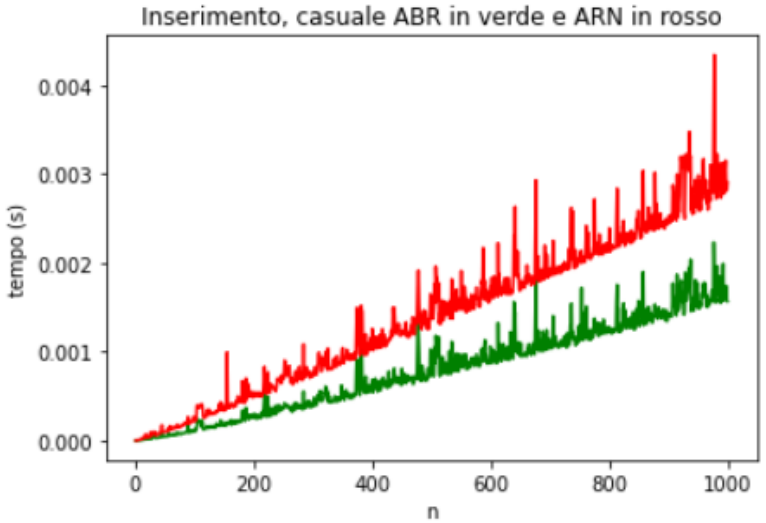

Figura 6: Grafici del tempo t (s) di ABR e ARN per Inserimento (caso random)

Osservazioni: Quanto visto nel caso peggiore non accade per il cosiddetto caso random, ovvero il caso medio degli alberi binari di ricerca: l'inserimento dei nodi in ordine casuale, come specificato nell'introduzione agli ABR, genera un albero approssimativamente bilanciato e quindi con tempi paragonabili a quelli degli ARN. Dai test sembrerebbe perfino che la scelta di un ABR sarebbe più conveniente rispetto a quella di un ARN, qualora si sappia a priori che i nodi non verranno aggiunti con un particolare ordinamento: un risultato probabilmente derivato dal fatto che un ARN, per essere bilanciato, ha bisogno di chiamare per ogni nodo la procedura `InsertFixup` che, pur avendo anch'essa tempo $O(lg(n))$, è comunque un insieme di operazioni aggiuntive che appesantiscono il programma.

5.4 Caso random di ricerca: Si mostra di seguito il tempo di esecuzione nel caso medio (ottenuto inserendo nodi in ordine casuale) dei due algoritmi su un albero con numero di nodi crescenti da 1 a 1000.

Tabella a confronto:

Numero di valori	Altezza ABR	Altezza ARN	Tempo ricerca casuale ABR (ms)	Tempo ricerca casuale ARN (ms)
0	-1	0	0.00067	0.00073
100	12	7	0.00096	0.00143
200	13	9	0.0013	0.00164
300	15	9	0.00128	0.00182
400	16	10	0.00115	0.00174
500	16	10	0.00189	0.00295
600	17	10	0.00176	0.00227
700	18	11	0.00147	0.00192
800	18	11	0.00221	0.00274
900	19	11	0.00133	0.00204
999	19	11	0.00268	0.0033

Figura 7: Tabella di ABR e ARN per Ricerca (caso random)

Grafici a confronto:

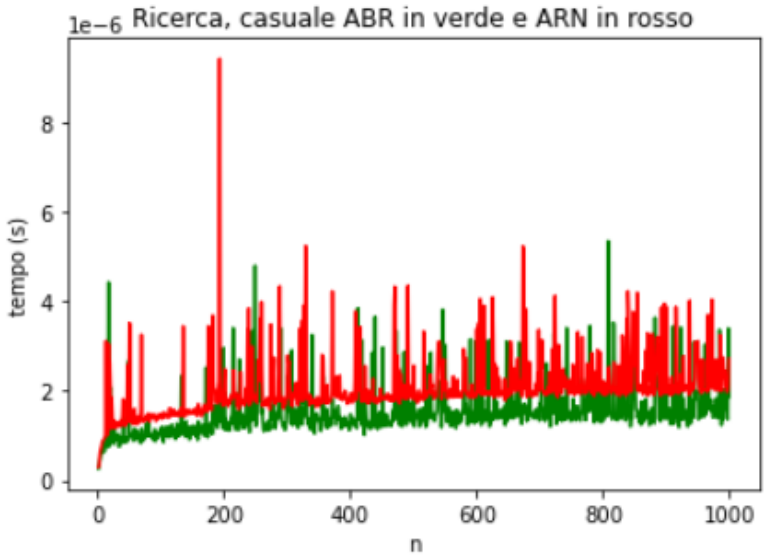

Figura 8: Grafici del tempo t (s) di ABR e ARN per Ricerca (caso random)

Osservazioni:

La ricerca nel caso random è invece circa equivalente alla ricerca nel caso peggiore di un ARN (riconfermando quindi il fatto che l'ARN non ha un caso peggiore perché è sempre bilanciato), ma questa volta anche per l'albero di ricerca binaria che, non avendo un solo nodo per ogni livello, ha un'altezza paragonabile a quella dell'albero rosso-nero.

5.5 Attraversamento: Si mostra di seguito il tempo di esecuzione nel caso medio (ottenuto inserendo nodi in ordine casuale) dei due algoritmi su un albero con numero

di nodi crescenti da 1 a 1000. L'attraversamento è stato applicato nel caso in cui l'ordine di inserimento dei nodi fosse casuale.

Tabella a confronto:

Numero di valori	Altezza ABR	Altezza ARN	Tempo attraversamento ABR (ms)	Tempo attraversamento ARN (ms)
0	-1	0	0.00072	0.0008
100	11	7	0.05562	0.09786
200	14	8	0.11172	0.19675
300	16	9	0.16777	0.29308
400	15	10	0.2526	0.37953
500	17	10	0.3112	0.52984
600	16	10	0.35786	0.58964
700	17	11	0.43266	0.69095
800	18	11	0.44506	0.80853
900	18	11	0.54475	0.88899
999	19	11	0.55112	0.98086

Figura 9: Tabella di ABR e ARN per Attraversamento

Grafici a confronto:

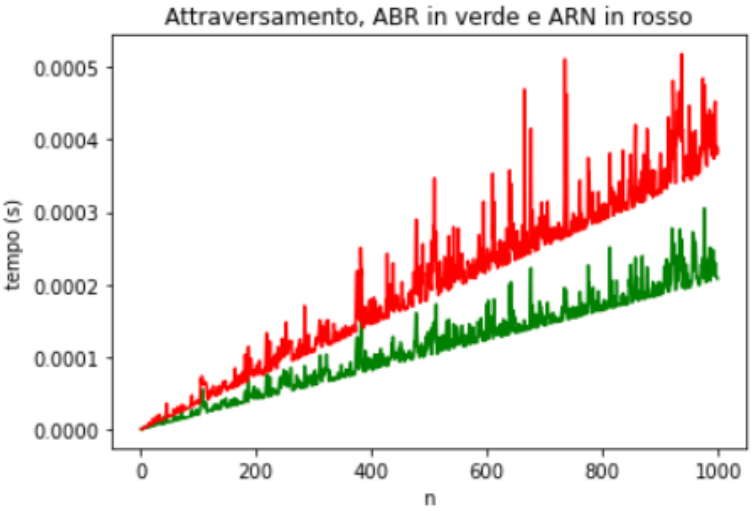

Figura 10: Grafici del tempo t (s) di ABR e ARN per Attraversamento

Osservazioni:

L'attraversamento inorder, che dovendo visitare tutti i nodi richiede $\Theta(n)$ a prescindere dall'altezza dell'albero, risulta come da aspettativa circa uguale sia per gli alberi binari di ricerca che per gli alberi rosso-neri ed è, a causa delle pochissime operazioni eseguite nel corpo della funzione, più veloce rispetto all'inserimento.

6 Conclusioni

Com'era intuibile, i risultati dei test ci portano a concludere che un albero rosso-nero è generalmente da preferire ad un semplice albero binario di ricerca, in quanto solitamente non si conosce l'ordine dei nodi che verranno inseriti, e non è perciò possibile

determinare a priori se nel tempo l'albero sarà bilanciato oppure no. Nel caso in cui ciò sia però noto, un ABR è equivalente ad un ARN, ed è anzi migliore nell'inserimento (data l'assenza della funzione ausiliaria `InsertFixup`). Anche se è stato analizzato un albero con al massimo 1000 nodi per il caso medio e peggiore, gli elementi utilizzati hanno permesso comunque di ottenere risultati sufficienti per le conclusioni cui si è giunti, dato che queste confermano le osservazioni teoriche fatte sugli alberi binari di ricerca e degli alberi rosso-neri. In generale gli alberi rosso-neri risultano migliori in quanto, grazie alle loro proprietà, consentono all'albero di essere più bilanciato rispetto ad un albero binario di ricerca, qualsiasi siano i nodi da inserire.