Confronto fra Alberi binari di ricerca e Alberi Rosso-Neri

Laboratorio di Algoritmi

Mattia Baroncelli

Ingegneria Informatica

Anno accademico 2022/2023

Indice

1 Introduzione

L'obiettivo di questa relazione è illustrare i principali vantaggi e svantaggi degli alberi binari di ricerca e degli alberi rosso-neri, mettendoli a confronto.
L'analisi si concentrerà prevalentemente sull' inserimento di elementi e sulla ricerca di questi ultimi all'interno delle due diverse strutture dati.

2 Strutture dati

Le due strutture dati si somigliano molto, nello specifico l'albero rosso nero è una struttura dati aumentata, proprio dell'albero binario di ricerca, di seguito analizzeremo meglio entrambi, valutandone a priori vantaggi e svantaggi.

2.1 Albero Binario di Ricerca

L'albero binario di ricerca è una struttura dati organizzata come dice il nome stesso come una vera e propria sorta di albero. E' costituita da numerosi nodi, che non sono altro che oggetti aventi i seguenti campi:

- Key: contiene i dati racchiusi nel nodo stesso
- Left: contiene il puntatore al figlio sinistro del nodo
- Right: contiene il puntatore al figlio destro del nodo
- Parent: contiene il puntatore al nodo padre

In un albero binario di ricerca è inoltre presente un nodo chiamato radice, che non presenta come padre alcun nodo

La proprietà più importante che i nodi di ogni albero binario di ricerca devono rispettare è la seguente:

- Ogni figlio sinistro ha il campo Key $\leq$ del proprio padre
- Ogni figlio destro ha il campo Key $\geq$ del proprio padre

2.2 Albero Rosso-Nero

L'albero rosso-nero è un'estensione del normale albero binario di ricerca, grazie all'aggiunta di un attributo "color" alla struttura dati.
L'attributo può prendere due valori, solitamente rappresentati in binario, che per praticità si fanno corrispondere ai colori rosso e nero. Attraverso l'aggiunta di questo attributo si riesce a garantire un stabilità della struttura dati, facendo in modo che l'albero risulti sempre bilanciato.

Oltre all'aggiunta dell'attributo "color" per poter essere bilanciato l'albero rosso-nero deve rispettare le seguenti linee guida:

- La radice è nera
- Ciascuna foglia (NIL) è nera
- Se un nodo è rosso allora entrambi i figli saranno neri.
- Ogni cammino semplice da un nodo ad una foglia sua discendente contiene lo stesso numero di nodi neri

Ciò che rende vantaggiosi gli alberi rosso-neri è la proprietà di avere al massimo un'altezza pari a $2\lg(n+1)$ dove n è il numero di nodi interni all'albero

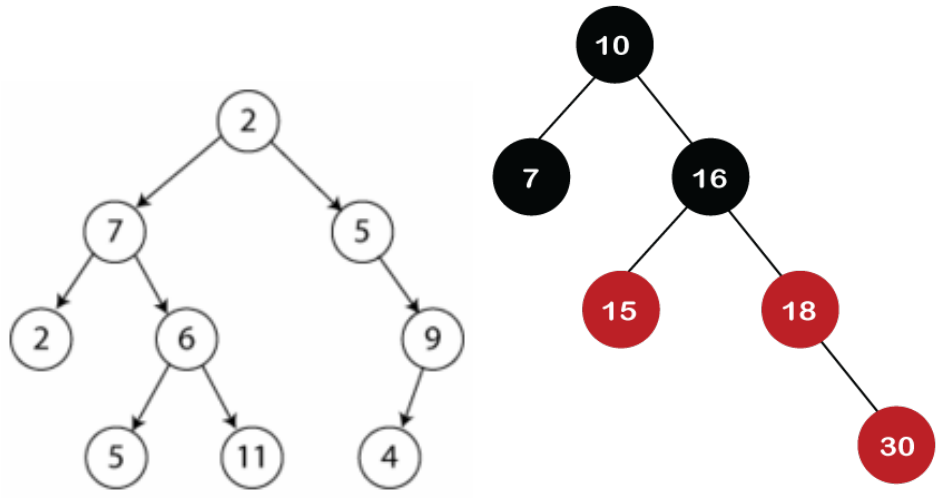

Figura 1: A sinistra un albero binario di ricerca a destra un albero rosso-nero.

3 Algoritmi e prestazioni

Gli algoritmi che operano sugli alberi binari di ricerca sono molteplici, quelli su cui intendo porre l'attenzione in questa analisi sono la ricerca e l'inserimento di un elemento all'interno dell'albero.

3.1 Algoritmo di ricerca

L'algoritmo di ricerca prende in input oltre all'albero, un valore x che sarà quello che andrà cercato all'interno dello stesso. L'algoritmo partendo dalla radice confronta il valore contenuto nella variabile x con il campo key del nodo preso in esame:

- x e il campo key sono uguali: in tal caso l'algoritmo ritorna vero
- x < del campo key: l'algoritmo si richiama ricorsivamente sul figlio sinistro del nodo corrente
- x > del campo key: l'algoritmo si richiama ricorsivamente sul figlio destro del nodo corrente

L'algoritmo è il medesimo sia per l'albero binario di ricerca che per l'albero rosso-nero.

3.2 Algoritmo di inserimento

L'algoritmo di inserimento prevede la modifica della struttura dati, aggiungendo nodi.
Nel caso dell' ABR questo non comporta grandi difficoltà, mentre nell'ARN obbliga il programmatore a dover tener conto di un ribilanciamento dell'albero. Questo aspetto non verrà analizzato nel profondo in questa relazione ma è comunque un aspetto da tenere in considerazione

L'algoritmo prende in input l'albero stesso e un nodo z da aggiungere all'albero. L'algoritmo agisce scendendo in fondo all'albero similmente a come viene fatto con l'operazione di ricerca, fino a quando non si arriva ad una foglia. A questo punto il nodo viene aggiunto come figlio sinistro o destro in base al valore dell'attributo key

3.3 Analisi delle prestazioni

Analizziamo adesso le prestazioni dei due algoritmi, con le nozioni date nel corso di Algoritmi e Strutture Dati. In questa analisi considereremo:

- Caso medio: un albero con valori dell'attributo Key generati pseudocasualmente
- Caso peggiore: un albero con valori dell'attributo Key ordinati in ordine non decrescente e inseriti all'interno dell'albero nello stesso ordine

Dai dati all'interno delle Tabelle 1 e 2 si può notare come nel caso medio i due algoritmi richiedano lo stesso tempo, che è proporzionale all'altezza dell'albero; sia per l'inserimento che per la ricerca. Ciò che invece rende evidente la convenienza dell'ARN rispetto all'ABR è l'analisi del caso peggiore. Infatti se per l'ABR il tempo è proporzionale al numero di nodi contenuti nell'albero, nell'ARN per entrambi gli algoritmi il tempo è proporzionale all'altezza dell'albero e visto che l'albero ha altezza logaritmica rispetto al numero dei nodi, questo rende di gran lunga conveniente implementare l'ARN dal punto di vista teorico.
Nonostante il caso medio non generi grandi discrepanze fra le due strutture dati, sarà comunque preso in analisi: sia per confermare il risultato teorico, ma anche per vedere come si comportano queste strutture dati nel caso di dimensioni molto grandi.

ABR	Caso medio	Caso peggiore
Ricerca	O(h)	O(n)
Inserimento	O(h)	O(n)

Tabella 1: Prestazioni algoritmi ABR

ARN	Caso medio	Caso peggiore
Ricerca	O(h)	O(h)
Inserimento	O(h)	O(h)

Tabella 2: Prestazioni algoritmi ARN

4 Documentazione del codice

Il primo passo fatto per arrivare a testare gli algoritmi è stato scrivere le due classi rappresentanti le due strutture dati. Di seguito il Diagramma UML delle due classi e le specifiche sui metodi di entrambe le classi

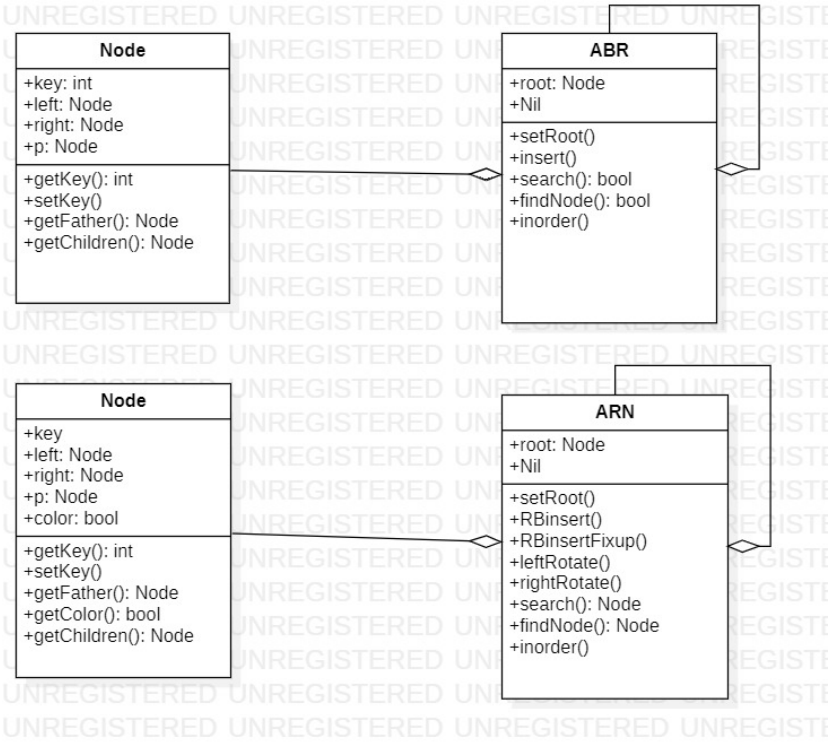

Figura 2: Diagramma UML classi ABR e ARN.

I file che contengono le seguenti classi sono due: ABR.py e ANR.py; entrambe contengono un attributo node, inoltre ognuna elle due contiene l'implementazione del relativo albero. Di seguito sono indicate le specifiche di ogni classe, per quanto riguarda attributi e metodi

4.1 Node

Implementa un nodo dell'ABR o dell'ARN.

4.1.1 Attributi

Ogni nodo è costituito dai seguenti attributi

- key: contiene la chiave contenuta in ogni nodo
- left: contiene il figlio sinistro del nodo
- right: contiene il figlio destro del nodo
- p : contiene il padre del nodo
- color: contiene un attributo booleano che specifica se il nodo è rosso (1) o nero (0) (solo ARN)

4.1.2 Metodi

Ogni nodo fa uso dei seguenti metodi

- getKey: restituisce il valore della chiave del nodo

- setKey: permette di modificare il valore della chiave del nodo
- getFather: restituisce il nodo padre
- getChildren: restituisce una lista contenente figlio sinistro e destro
- getColor: restituisce il colore del nodo (solo ARN)

4.2 ABR

Implementa l'albero binario di ricerca

4.2.1 Attributi

L'ABR è costituito dai seguenti attributi

- root: contiene il nodo attribuito alla radice
- Nil: non è un attributo vero e proprio, verrà settato a valori diversi e indica il valore nullo di un particolare attributo del nodo

4.2.2 Metodi

Ogni ABR fa uso dei seguenti metodi

- setRoot: permette di settare il valore della chiave della radice
- insert: inserisce un nuovo valore all'interno dell'ABR
- search: chiama la funzione findNode facendo risultare trasparente la ricerca
- findNode: ricerca un valore all'interno dell'albero e restituisce vero se lo trova
- inorder: visita i nodi dell'albero in ordine crescente di chiave

4.3 ARN

Implementa l'albero rosso-nero, estendendo la lista degli attributi di ABR e aggiungendo metodi

4.3.1 Attributi

L'ANR è costituito dai seguenti attributi

- root: contiene il nodo attribuito alla radice
- Nil: non è un attributo vero e proprio, verrà settato a valori diversi e indica il valore nullo di un particolare attributo del nodo

4.3.2 Metodi

Ogni ANR fa uso dei seguenti metodi

- setRoot: permette di settare il valore della chiave della radice
- RBinsert: inserisce un nuovo valore all'interno dell'albero (l'albero non è ancora bilanciato)
- RBinsertFixup: viene chiamata da RBinsert e bilancia l'albero durante l'inserimento di un nuovo valore
- leftRotate: esegue una rotazione sinistra dell'albero
- rightRotate: esegue una rotazione destra dell'albero
- search: chiama la funzione findNode facendo risultare trasparente la ricerca
- findNode: ricerca un valore all'interno dell'albero e restituisce vero se lo trova
- inorder: visita i nodi dell'albero in ordine crescente di chiave

4.4 Discussione delle scelte implementative

Il codice sopra proposto cerca di rimanere fedele il più possibile allo pseudocodice fornito dal libro di testo, nonostante questo intendo soffermarmi su alcune scelte. La prima che vorrei discutere riguarda il non vincolare le due strutture dati ad una relazione di ereditarietà. Questa scelta è stata dettata principalmente dal fatto che il metodo principale alla base di questa analisi, ovvero l'inserimento, ha molte differenze fra le due strutture dati, tanto da farle sembrare due strutture dati diverse. Pertanto nonostante i nodi degli alberi differiscono per solo un attributo di differenza ho preferito creare una nuova struttura dati (da zero) per rappresentare l'ARN lasciando la struttura degli altri metodi quasi inalterata.

5 Descrizione degli esperimenti

L'obiettivo dei test effettuati è avere un riscontro dei concetti teorici studiati nella pratica.
L'analisi fatta in questa relazione si concentrerà sull'inserimento e sulla ricerca di determinati valori all'interno delle due strutture dati.
Come detto nella sezione 3.3 mi concentrerò su alberi costruiti inserendo valori ordinati in ordine strettamente crescente (caso peggiore) e inserendo valori random (caso medio).

Per quanto riguarda l'implementazione del caso peggiore l'ho fatta attraverso un semplice ciclo "for" che ad ogni iterazione inserisce il valore dell'indice all'interno dell'albero.

Per l'implementazione del caso medio ho creato una lista di valori ordinati e tramite la funzione shuffle della libreria random ho randomizzato la stessa per poi inserire i valori all'interno dell'albero sempre tramite un ciclo for. E' da notare quindi che la randomizzazione non permette ripetizioni di valori uguali all'interno dell'albero, poichè la lista dalla quale vengono aggiunti contiene inizialmente numeri tutti diversi

Per misurare il tempo trascorso dall'inizio alla fine di un esperimento ho utilizzato la funzione defaulttimer dalla libreria timeit

La nostra analisi si concentrerà sul confronto fra alberi con un diverso numero di nodi al proprio interno e sul confronto a parità di nodi fra ABR e ARN. In particolare analizzeremo alberi con 20 elementi, con 200 elementi e con 20000 elementi sia nel caso peggiore che nel caso medio.

5.1 Esperimento sull'inserimento

Concentriamoci adesso sull'inserimento di valori tutti diversi all'interno dell'albero analizzando come scritto nella sezione precedente.

5.1.1 Inserimento di 20 elementi

Il caso dell'inserimento con 20 elementi è un caso interessante da studiare per verificare se anche con un numero di nodi relativamente piccolo, l'ARN sia da preferirsi all'ABR. Per verificarlo andiamo a fare due test:
Nel primo test inseriamo 20 numeri (da 1 a 20) randomicamente all'interno dei nodi dei due alberi
Come possiamo notare nei grafici sottostanti l'inserimento di 20 valori random all'interno dei due alberi non crea molte differenze a livello di tempo computazionale. Questo è dovuto al fatto che nella maggior parte dei casi l'ABR sarà bilanciato anche senza le modifiche fatte nell'ARN (è questo il caso indicato nel grafico). E nel caso di cammini più lunghi da percorrere in media questi si verificheranno di rado e con una lunghezza mai superiore ai 20 nodi Ovviamente questo discorso si adatta al caso di elementi inseriti casualmente nell'albero.

Per quanto riguarda il caso di inserimento di elementi ordinati la situazione è ben diversa.
Nonostante i nodi inseriti siano solo 20 c'è una grande differenza fra il grafico dell'ABR e quello dell'ARN. Nell'ABR infatti i primi 10 elementi vengono inseriti dando vita ad una retta nel grafico, mentre dal decimo in poi sembra avere un andamento simile ad una curva esponenziale. Questo comportamento è causato dal fatto che l'albero ha al suo interno un solo cammino disponibile ed è obbligato ogni volta a compierlo fino in fondo per poter aggiungere un elemento; e se per i primi

elementi il cammino non è così lungo, per i successivi il tempo per l'inserimento di un nuovo elemento diventa esponenziale. Per quanto riguarda L'ARN grazie a questo esperimento si può notare un fatto molto interessante. L'albero è costretto continuamente a bilanciarsi, creando nel grafico sezioni in cui cresce molto rapidamente a sezioni dove la retta ha un coefficiente angolare minore.
L'analisi del tempo effettivo impiegato per terminare l'algoritmo non si adatta bene a questo (e al prossimo) caso, poichè si parla di tempi molto brevi. Preferisco pertanto rimandarla all'inserimento di 20000 elementi

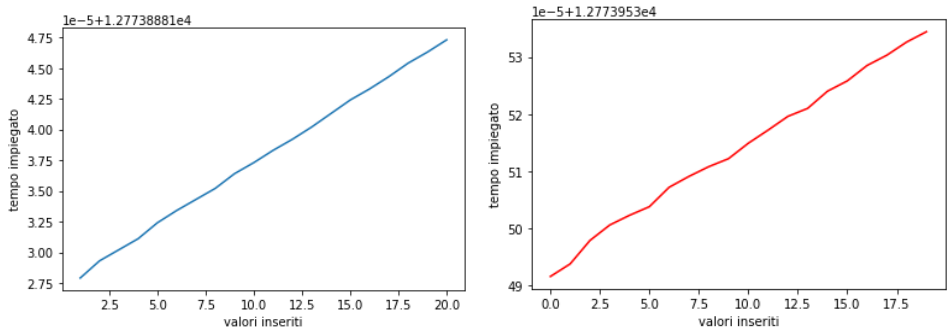

Figura 3: Inserimento di 20 elementi randomici: ABR in blu, ARN in rosso

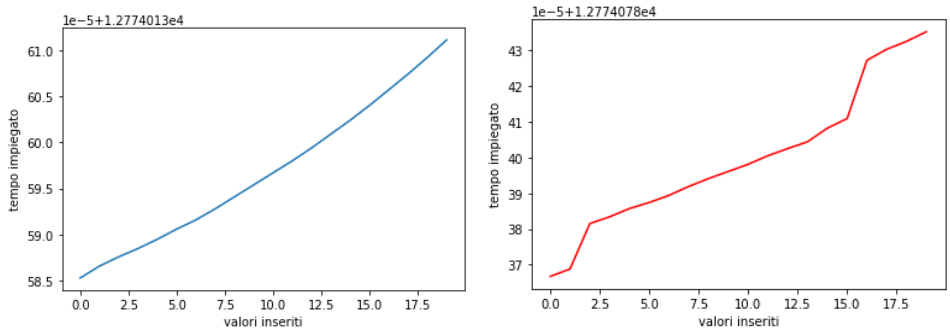

Figura 4: Inserimento di 20 elementi ordinati in ordine strettamente crescente: ABR in blu, ARN in rosso

5.1.2 Inserimento di 200 elementi

Proviamo adesso ad inserire un numero di elementi 10 volte più grande del caso precedente.
L'interesse stavolta si ha non solo sul confronto fra le due strutture dati ma anche quello fra i due casi sulla stessa struttura dati.
Sull'inserimento di valori pseudocasuali, l'aumentare del numero di nodi fa sì le prestazioni delle due strutture dati si assomiglino sempre di più. Tanto che a seconda dell'esecuzione del test l'inserimento su una struttura dati impiegava meno tempo rispetto all'altra e viceversa.

Nel caso di inserimento di valori ordinati, invece, più si aumenta il numero di nodi presenti nell'albero e più le due strutture dati differiscono. Per quanto riguarda il tempo effettivo non si notano grandi differenze, poichè entrambe impiegano una frazione di secondo per eseguire tutto l'algoritmo. La differenza si nota nella computazione dell'algoritmo; infatti se il grafico dell'ARN non differisce molto dal caso precedente, quello dell'ABR ha un andamento esponenziale fin quasi da subito. Impiegando quindi a maggior parte del tempo per inserire gli ultimi valori dell'albero. Tempo che comunque risulta al pari dell'ARN poichè questo impiega il tempo a bilanciarsi.

5.1.3 Inserimento di 20000 elementi

Avendo confermato ciò che avevamo visto nella teoria, proviamo ad aumentare di molto il numero di nodi da inserire, inserendone 20000.

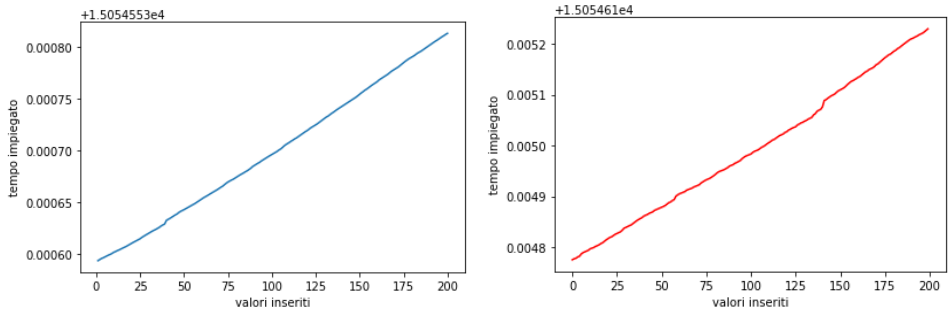

Figura 5: Inserimento di 200 elementi randomici: ABR in blu, ARN in rosso

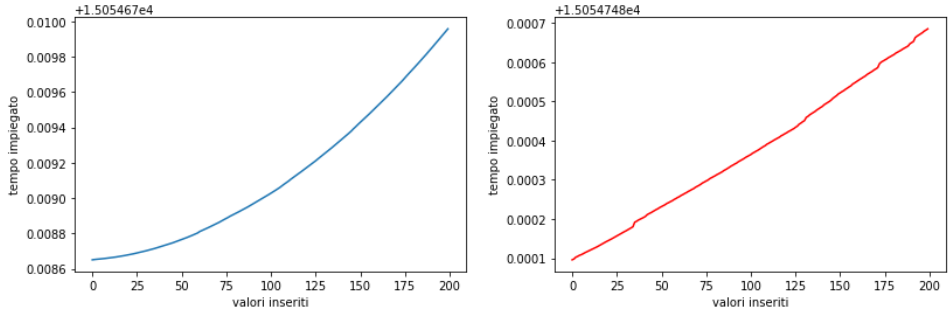

Figura 6: Inserimento di 200 elementi ordinati in ordine strettamente crescente: ABR in blu, ARN in rosso

Nel caso di valori inseriti randomicamente il tempo medio impiegato di entrambi non varia moltissimo da quello osservato nell'inserimento di 200 elementi, ma come abbiamo visto precedentemente questo dipende anche da che numeri vengono assegnati dalla macchina e quindi è un caso che ci interessa ma relativamente.

Molto più interessante è il caso dei valori inseriti in ordine crescente, in questo caso come si può evincere dalla Figura 8 la differenza fra le due strutture dati è enorme. Se per l'albero rosso-nero fare questa operazione non richiede neanche un secondo, per l'albero binario di ricerca richiede sempre più di 10 secondi.
Di seguito ho inserito una media dei tempi impiegati dall'algoritmo (in secondi) fatta su 15 misurazioni.
Anche i grafici sottostanti indicano questo, il comportamento dell'ARN tende a rimanere sempre lo stesso, mentre l'ABR più elementi inserisce e più tempo impiega per inserire i successivi. In particolare per inserire gli ultimi 1000 valori impiega quasi 4 secondi.

	Elementi pseudocasuali	Elementi ordinati
ABR	0.053 s	12.732 s
ARN	0.059 s	0.080 s

Tabella 3: Tempo effettivo impiegato per inserire 20000 valori all'interno delle due strutture dati (Media di 15 misurazioni)

5.2 Esperimento sulla ricerca

Per quanto riguarda la ricerca l'esperimento che intendo fare è ricercare un elemento casuale all'interno delle due strutture dati sia nel caso che sia stato costruito aggiungendo valori randomicamente, sia aggiungendo valori ordinati. L'analisi intende verificare il caso peggiore dell'ABR andando a ricercare l'ultimo elemento inserito all'interno della struttura
E' da notare che la ricerca è implementata tramite il metodo ricorsivo e non quello iterativo.

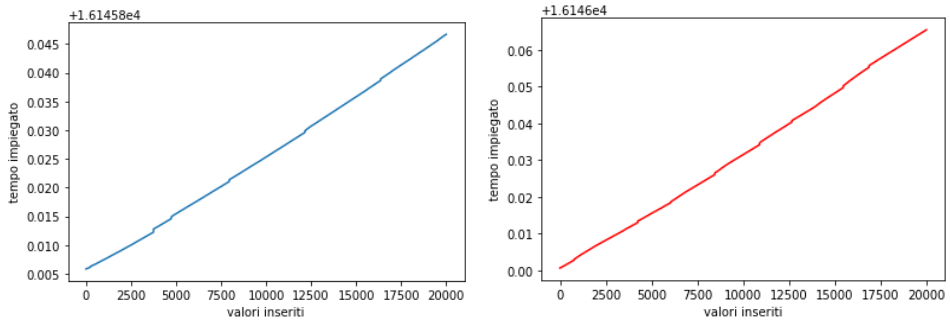

Figura 7: Inserimento di 20000 elementi ordinati pseudocasualmente: ABR in blu, ARN in rosso

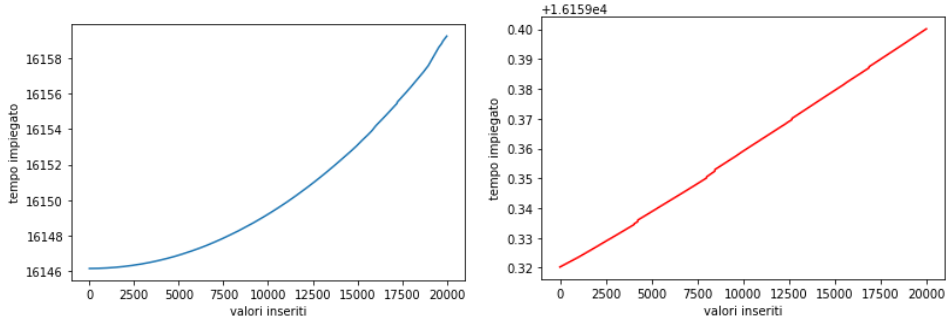

Figura 8: Inserimento di 20000 elementi ordinati in ordine strettamente crescente: ABR in blu, ARN in rosso

Le tabelle sottostanti indicano il tempo impiegato per la ricerca dell'ultimo elemento inserito nell'albero nei seguenti casi:

- L'albero è stato creato inserendo valori in ordine pseudocasuale al suo interno
- L'albero è stato costruito inserendo valori in ordine strettamente crescente al suo interno.

Si noti che non ho potuto replicare il numero di nodi precedentemente fatto (20000) poichè il programma si bloccava in seguito alle troppe chiamate ricorsive fatte.

	N=200	N=2000
ABR	$5.19\ 10^{-6}$	$1.15\ 10^{-5}$
ARN	$2.70\ 10^{-6}$	$4.69\ 10^{-6}$

Tabella 4: Tempo effettivo per ricercare l'ultimo elemento in un ABR con elementi casuali al suo interno

(Media di 15 misurazioni)

	N=200	N=2000
ABR	$3.67\ 10^{-5}$	$6.20\ 10^{-4}$
ARN	$2.09\ 10^{-6}$	$4.40\ 10^{-6}$

Tabella 5: Tempo effettivo per ricercare l'ultimo elemento in un ABR con elementi ordinati al suo interno

(Media di 15 misurazioni)

Dai dati illustrati nelle Tabelle 4 e 5 si può notare come nel caso dell'albero costruito con valori pseudocasuali i tempi si assomiglino molto nel caso di 200 nodi, con un leggero vantaggio per l'ARN e che cambino nel caso dell'albero formato da 2000 nodi di poco meno di un ordine di grandezza.

La cosa diventa ancora più interessante andando ad osservare il caso peggiore per l'ABR ovvero la ricerca di un valore quando il percorso da fare è massimo. In questo caso l'ARN non subisce grandi cambiamenti, se non di poco dato dalla casualità dei numeri. Mentre sia nel caso di 200 che in quello di 2000 nodi l'ABR impiega dieci volte tanto per eseguire la ricerca rispetto al caso pseudocasuale. Inoltre la differenza fra ABR e ARN nel caso estremizzato di 2000 nodi diventa di ben 2 ordini di grandezza.

6 Conclusioni finali

Gli esperimenti appena fatti hanno messo a confronto queste due strutture dati apparentemente molto simili facendone uscire fuori pregi e difetti, in praticolare:

- L'inserimento nel caso di nodi inseriti in ordine pseudocasuale se fatto su un numero di nodi relativamente piccolo è conveniente farlo su un ABR, poichè non necessita di bilanciarsi come l'ARN. Al crescere del numero di nodi, invece conviene utilizzare l'ARN.
- Sempre sull'inserimento ma nel caso di nodi inseriti in ordine crescente abbiamo visto che il tempo impiegato dall'ARN rimane sempre lineare, mentre anche per valori non molto grandi l'ABR ha una curva che ricorda la curva di una esponenziale.
- Anche per la ricerca è molto importante tenere in considerazione la forma dell'albero che si prende in considerazione per l'ABR.
- Nel caso di un numero di elementi molto grande per la ricerca conviene utilizzare l'ARN indipendentemente dal modo in cui sono stati inseriti i nodi (almeno che non siano stati inseriti in modo da bilanciare perfettamente l'albero)

7 Specifiche hardware e software utilizzati

Processore 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz 2.42 GHz

RAM installata 8,00 GB (7,70 GB utilizzabile)

Sistema Operativo Windows 11

Tipo sistema Sistema operativo a 64 bit, processore basato su x64

Ambiente di sviluppo utilizzato: Spyder IDE

Editor LaTeX: Overleaf

8 Bibliografia

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein (2009) Introduzione agli algoritmi e strutture dati Terza edizione, McGraw Hill.
- Slide Algoritmi e Strutture Dati (corso 2021/22)
- Slide Laboratorio di Algoritmi (corso 2021/22)