Laboratorio di Algoritmi e Strutture dati

Alessia Ludovica Bruno
Matricola: 7047552
alessia.bruno2@stud.unifi.it

Anno accademico 2022/2023

Università degli Studi di Firenze
Dipartimento di Ingegneria dell'Informazione

Corso Principale: Algoritmi e Strutture Dati
Docente: Simone Marinai

Indice

Elenco delle figure

1 Introduzione generale

1.1 Esercizio assegnato

Per il superamento della parte di Laboratorio di Algoritmi e Strutture Dati ho svolto il seguente esercizio:

- B : Statistiche d'ordine dinamiche - varie implementazioni

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 utilizzata per i test è un Laptop Lenovo, modello ideapad 700-15ISK con Sistema operativo Windows 10 Home versione 21H2, 64 bit. L'Hardware del dispositivo è il seguente:

- **CPU** : Intel Core i7-6700HQ CPU 2.60GHz
- **RAM** : 8GB
- **SSD** : 465GB

Il linguaggio di programmazione utilizzato sarà Python e la piattaforma in cui il codice è stato scritto ed eseguito è l'IDE **Spyder**. La stesura di questo testo è avvenuta tramite l'utilizzo dell'editor online **Overleaf**.

Statistiche d'ordine dinamiche

B: Statistiche d'ordine dinamiche

- Vogliamo confrontare varie implementazioni di statistiche d'ordine dinamiche:
  1. Con lista ordinata
  2. con ABR senza attributo size
  3. Come visto a lezione
- Per fare questo dovremo:
  - Scrivere i programmi Python (no notebook) che:
    - implementino quanto richiesto
    - eseguono un insieme di test che ci permettano di comprendere vantaggi e svantaggi delle diverse implementazioni
  - Svolgere ed analizzare opportuni esperimenti
  - Scrivere una relazione (in LaTeX) che descriva quanto fatto
  - Nota: le strutture dati devono sempre essere implementate nel progetto; non si possono utilizzare librerie sviluppate da altri o copiare codice di altri

2 Spiegazione teorica dell'esercizio

2.1 Introduzione

L'esercizio richiede di implementare le statistiche d'ordine dinamiche non solo nel classico metodo, ovvero con un albero rosso nero con in aggiunta l'attributo size, ma anche con un comune albero binario di ricerca e con una lista collegata da puntatori, in particolare che sia già ordinata (D'ora in avanti mi riferirò a questa struttura dati come lista ordinata). Fatto ciò, si vuole misurare la complessità computazionale delle operazioni tipiche delle statistiche d'ordine dinamiche, **Os-Rank** e **Os-Select**, su ciascuna delle tre implementazioni, al fine di decidere quale delle tre strutture dati convenga di più usare.

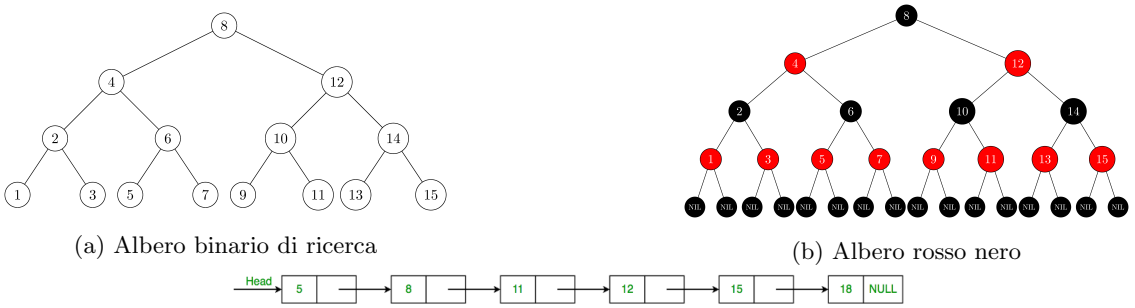
2.2 Concetti teorici sulle strutture dati

Per eseguire i test sopra citati avremo bisogno di sfruttare alcune operazioni di base come **inserimento** e **ricerca**, oltre appunto a **Os-Rank** e **Os-Select**. In questo paragrafo verrà fornita una breve spiegazione su come queste operazioni vengono implementate sulla struttura dati, insieme ad una descrizione sintetica sulle caratteristiche degli alberi e della lista. Cominciamo da quest'ultima.

- **Albero binario di ricerca**: si tratta di una struttura dati, in particolare di un albero dove ogni nodo ha al massimo due figli. Esso viene costruito rispettando la seguente proprietà: chiave del figlio destro ¡ chiave in cui mi trovo ¡ chiave figlio sinistro. Per ogni albero devo disporre dei seguenti campi: un puntatore alla radice (il primo nodo), un puntatore al figlio sinistro, un puntatore al figlio destro, un puntatore a padre (Perciò il puntatore al padre della radice punterà a niente) ed infine la chiave del nodo, ovvero il valore che esso contiene.
- **Albero rosso nero**: Ha le stesse proprietà di un albero binario di ricerca, ma ogni nodo possiede un attributo booleano "color" che, se rispetta le seguenti proprietà, garantisce che non ci siano alberi degeneri (Un esempio di albero degenere può essere un albero con solo figli sinistri, che diventerebbe una lista puntata...)
  1. Ogni nodo è rosso o nero
  2. La radice è nera
  3. Ogni foglia è nera
  4. Se un nodo è rosso, allora entrambi i suoi figli sono neri

5. Tutti i camini da un nodo alle foglie contengono lo stesso numero di nodi neri

- **Lista ordinata**: Un semplice "elenco" di elementi sistemati in ordine crescente. Ogni nodo della lista è costituito dalla chiave e da un puntatore all'elemento successivo (Il puntatore a "next" dell'ultimo elemento punta a null)



(a) Albero binario di ricerca

(b) Albero rosso nero

(c) Lista ordinata con puntatori

Figura 1: Tipologie di alberi

Adesso proseguiamo con la spiegazione di alcune operazioni utili che serviranno ai fini degli esperimenti

- Inserimento
  - Albero Binario di Ricerca: Inizio dalla radice dell'albero ed imposto delle variabili che tengano traccia del nodo padre e del nodo figlio, quindi discendo lungo l'albero, spostandomi via via o a sinistra o a destra, a seconda se la chiave da inserire è rispettivamente minore o maggiore della corrente. Infine aggiusto i puntatori.
  - Albero Rosso Nero: Il procedimento è il medesimo, con la differenza che il colore del nodo inserito viene impostato a "rosso" e che viene successivamente chiamata la funzione fixup al fine di preservare le proprietà dell'albero rosso nero.
  - Lista Ordinata: Si inserisce l'elemento al posto corretto nella lista, spostando di un indice tutti gli elementi maggiori di quello nuovo ed aggiustando i puntatori.
- Ricerca
  - Albero Binario di Ricerca: Mi muovo lungo l'albero con la stessa logica dell'inserimento e ad ogni nodo controllo che sia uguale a quello che sto cercando. In caso affermativo ritorno un puntatore al suddetto nodo, altrimenti, arrivati in fondo all'albero senza riscontrare successi, ritorno un puntatore vuoto.
  - Albero Rosso Nero: Come sopra
  - Lista Ordinata: L'algoritmo di ricerca binaria aiuta a trovare l'elemento richiesto senza bisogno di scorrere tutta la lista. Non avendone avuto bisogno ai fini dell'esperimento non è stato implementato nel codice e perciò non ci dilungheremo.

Infine, le operazioni di cui vogliamo studiare la complessità

- Os-Rank
  - Albero Binario di Ricerca: Si vuole il rango di un nodo con data chiave. Il rango è per definizione, il numero di elementi presenti nel sotto albero con radice in quel nodo più il nodo stesso. É stato pensato un tipo di approccio che sfrutta un'operazione molto comune, l'attraversamento **in order**. Nello specifico, per ogni nodo di cui vogliamo determinare il rango, si scorre l'albero fino a trovare il suddetto nodo, incrementando via via la variabile in cui si salva il rango.
  - Albero Rosso Nero: Come sopra, con la differenza che, per definizione, le statistiche d'ordine dinamiche non sono altro che alberi rosso neri dove ad ogni nodo, oltre ai classici attributi left, right, parent e color si aggiunge anche size. Perciò non sarà necessario attraversare il sotto albero ad ogni iterazione per calcolarsi il rango, in quanto la dimensione sarà informazione fornita direttamente dal nodo che stiamo visitando.

  - Lista Ordinata: Nel caso della lista, una volta fornito il nodo di cui si vuole il rango sarà necessario contare gli elementi uno ad uno scorrendola, nel caso peggiore, fino alla fine.

- Os-Select
  - Albero Binario di Ricerca: Si vuole il puntatore al nodo che contiene l' n-esima chiave più piccola nel sotto albero con radice nel nodo dato. Come sopra, si dovrà scorrere tutto il sotto albero tramite l'attraversamento **in order**.
  - Albero Rosso Nero: Il ragionamento è il medesimo; la size del nodo ci viene fornita dalla struttura dati stessa e perciò non va ricalcolata.
  - Lista Ordinata: Come sopra.

2.3 Complessità d'ipotesi

Per gli Alberi binari di ricerca le operazioni di inserimento richiedono un tempo di $O(h)$, dove h è l'altezza dell'albero ed è pari a log n nel caso migliore, ovvero quello di un albero bilanciato, mentre è pari a $O(n)$ nel caso peggiore, quello di un albero degenere (o catena lineare di elementi). Lo stesso vale per l'albero rosso nero, con la differenza che non avremo mai il caso peggiore, proprio perchè per definizione deve rispettare le proprietà sopra elencate. Nella lista invece, l'inserimento ha un caso ottimo di $O(1)$ se trovo immediatamente il punto in cui inserire l'elemento, in quel caso si tratta solo di aggiornare i puntatori. Nel caso peggiore, quando la lista deve essere attraversata quasi tutta per trovare il punto di inserimento, il tempo stimato è $O(n)$ dove n è il numero di elementi della lista.

Giunti a questo punto, sappiamo che il tempo stimato per le operazioni **Os-Select** e **Os-Rank** è $O(logn)$ per le statistiche d'ordine dinamiche implementate con alberi rosso neri, ma per le altre due implementazioni possiamo solamente fare una stima.

Nel caso dell'albero binario di ricerca, dovendo scorrere il sotto albero per ogni nodo, potrebbe essere necessario $O(n)$ nel caso peggiore (lista concatenata) per entrambe le operazioni.

Per la lista ordinata invece, sia per **Os-Rank** che per **Os-Select** potremmo avvicinarci a $O(n)$ nel caso peggiore, ovvero quello in cui dobbiamo scorrere tutta la lista di n elementi.

	Inserimento	Ricerca	Os-rank	Os-Select
ABR	$O(h)$	$O(h)$	$O(n)$	$O(n)$
RB	$O(h)$	$O(h)$	$O(logn)$	$O(logn)$
Lista ordinata	$O(n)$	$O(n)$	$O(n)$	$O(n)$

Figura 2: Tabella riassuntiva delle complessità stimate

Come accennato in precedenza le statistiche d'ordine dinamiche nascono con la necessità di voler aggiungere informazioni ad una struttura dati nota preservando i tempi di computazione conosciuti. Per questo per aumentare le strutture dati normalmente si seguono queste poche e semplici regole.

1. Scegliere una struttura dati sottostante (Alberi RN)
2. Determinare informazioni aggiuntive da gestire (size del nodo)
3. Verificare che si possano gestire le informazioni aggiuntive per le operazioni esistenti sulla struttura dati con analoghi tempi di esecuzione (tenere aggiornato size durante inserimenti e cancellazioni)
4. Sviluppare nuove operazioni (OS-Select e OS-Rank)

Per gestire size in inserimento in modo efficiente incremento size di ogni nodo visitato mentre scendo nell'albero; dopotutto il nuovo nodo sarà un discendente di ogni nodo già visitato. Nelle rotazioni (Nel fixup) basta aggiungere in fondo allo pseudo codice y.size ← x.size e x.size ← x.left.size + x.right.size + 1, spendendo solamente O(1) per aggiornarlo ad ogni rotazione.

3 Descrizione dei metodi implementati

Di seguito un diagramma UML allo scopo di mostrare com'è stato strutturato il codice. Tutte e tre le tipologie di nodi hanno una relazione di aggregazione con i rispettivi alberi. Vi è anche una relazione di aggregazione tra **RBNode** e **Node** con molteplicità 3, dato che entrambe hanno all'interno le istanze del nodo padre, del figlio destro e del figlio sinistro.

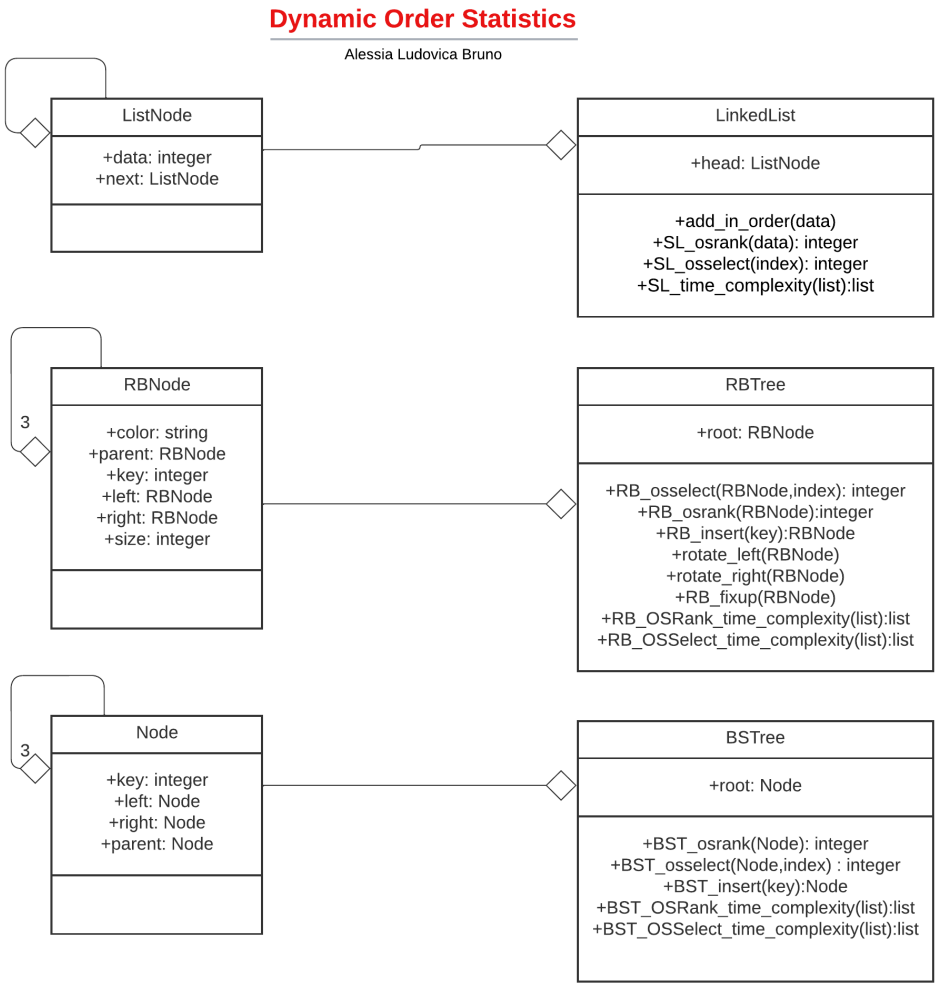


Figura 3: UML realizzato con Lucidchart

Di seguito le funzionalità dei metodi divisi per struttura dati

- **Lista ordinata**
  - **add_in_order** inserisce l'elemento passato nella lista in modo da preservare l'ordine
  - **SL_osrank** Funzione os-rank implementata per la lista
  - **SL_osselect** Funzione os-select implementata per la lista
  - **SL_OSRank_time_complexity** Misura la complessità di os-rank per la lista ordinata, stampandone il grafico.
  - **SL_OSSelect_time_complexity** Misura la complessità di os-select per la lista ordinata, stampandone il grafico.
- **Albero rosso nero**
  - **RB_insert** inserisce l'elemento passato nell'albero rosso nero
  - **RB_fixup** chiamato dall'inserimento per fare in modo che dopo l'aggiunta di un nuovo nodo vengano rispettate le proprietà dell'albero.

  - **rotate_left** rotazione per ribilanciare l'albero, metodo chiamato nel fixup.
  - **rotate_right** rotazione per ribilanciare l'albero, metodo chiamato nel fixup.
  - **RB_osrank** Funzione os-rank implementata per l'albero rosso nero
  - **RB_osselect** Funzione os-select implementata per l'albero rosso nero
  - **RB_OSRank_time_complexity** Misura la complessità di os-rank per l'albero rosso nero, stampando il grafico.
  - **RB_OSSelect_time_complexity** Misura la complessità di os-select per l'albero rosso nero, stampando il grafico.

- **Albero binario di ricerca**

  - **BST_osselect** Funzione os-select implementata per l'albero binario di ricerca
  - **BST_osrank** Funzione os-rank implementata per l'albero binario di ricerca
  - **BST_insert** Inserisce il nodo passato nell'albero binario di ricerca
  - **BST_OSRank_time_complexity** Misura la complessità di os-rank per l'albero binario di ricerca, stampando il grafico.
  - **BST_OSSelect_time_complexity** Misura la complessità di os-select per l'albero binario di ricerca, stampando il grafico.

Infine abbiamo la funzione **search** che è fuori dalle classi perchè mi è utile per cercare i nodi in entrambe le tipologie di albero.

Il programma principale crea una lista di 2000 interi randomici compresi nell'intervallo tra 0 e 4000, passandoli alle funzioni incaricate di calcolare la complessità in modo tale da avere la stessa tipologia di elementi per ogni struttura dati. Inoltre ha il compito di creare dei grafici con gli elementi sopra citati e le liste dei tempi ritornate dalle funzioni in modo tale da confrontarli.

4 Esperimenti

4.1 Descrizione degli esperimenti

Per calcolare la complessità ci serviremo della libreria Python "matplotlib" che con la sua funzione plot(x,y) permette di creare grafici passandogli due liste di parametri. Per tutte e tre le strutture dati la logica è stata la seguente: provare varie lunghezze di lista, da 10 fino a 2000 elementi, con un passo di dieci (Si provano quindi 200 lunghezze diverse di lista) e per ogni lunghezza salvare in un array il tempo necessario a svolgere la funzione richiesta. Viene fatta poi la media tra i tempi ottenuti, sommandoli e dividendoli per il numero di ripetizioni effettuato. Di seguito un pezzo di codice generico che mostra come sono strutturate le funzioni per calcolare a complessità.

```python
def time_complexity(numbers):

        struttura = StrutturaDati()
        size=2000
        step=10
        num_tests = 10
        times = []

        for n in numbers:
            struttura.add(n)

        for s in range(step, size + step, step):
            subset_numbers = numbers[:s]
            test_times = []
            for i in range(num_tests):
                total_time = 0
                for n in subset_numbers:
                    start_time = timeit.default_timer()
                    struttura.FunzioneDaTestare(n)
                    elapsed_time = timeit.default_timer() - start_time
                    total_time += elapsed_time
                test_times.append(total_time)
            avg_time = statistics.mean(test_times)
            times.append(avg_time)
            if(s==200 or s==2000):
                print("Os Rank per struttura con",s,"elementi",format(avg_time, '.6f'))


        plt.title('Time Complexity of OSRank')
        plt.xlabel("Size")
        plt.ylabel("Time required")
        plt.plot(range(step,size+step,step), times)
        plt.show()

        return times
```

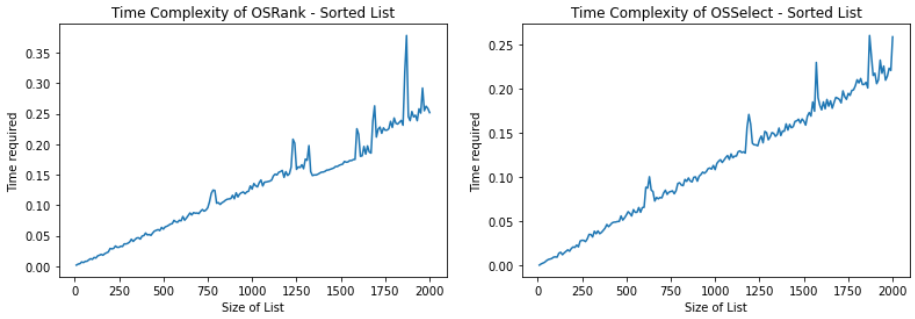
Figura 4: Struttura generale delle funzioni

4.2 Risultati sperimentali

Di seguito verrà seguito lo stesso ordine usato nel codice

4.2.1 Lista ordinata

Come accennato, creo una lista chiamata "linked", imposto un range di lunghezze che va da 10 a 2000 con passo 10 (controllo quindi 200 liste diverse) e per ogni dimensione di lista, la riempio con elementi casuali e calcolo il rango per ognuno di questi, misurando il tempo di esecuzione. Per ottenere un grafico il più reciso possibile ho svolto i calcoli più di una volta tramite un apposito ciclo for e infine ho calcolato la media. Lo stesso vale per Os-select. Il risultato è stato il seguente



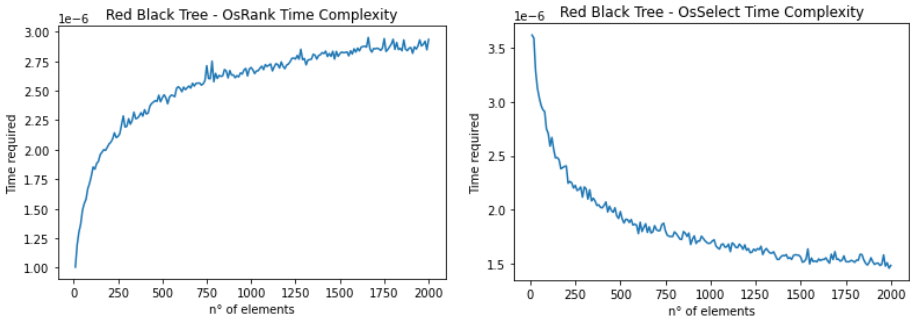
(a) Os-Rank (b) Os-Select

Figura 5: Complessità stimate sulla lista ordinata

Come premesso, la complessità di Os-rank ha chiaramente un andamento lineare $O(n)$, così come per Os-select.

4.2.2 Albero rosso nero

Anche qui creo un albero, imposto la dimensione massima a 2000 elementi, il passo è sempre di 10, quindi anche qui controllo 200 alberi di dimensioni diverse. Uso la funzione search per cercare il nodo con la chiave scelta dall'algoritmo e restituirne il puntatore alla funzione rank. Il ragionamento è stato il medesimo anche per select, con la differenza che ho dovuto aggiungere qualche linea di codice in più per scegliere un indice casuale "index" che rappresenterebbe la i-esima chiave più piccola che si sta cercando. Anche qui sono state fatte le medie dei grafici. Il risultato è stato il seguente:



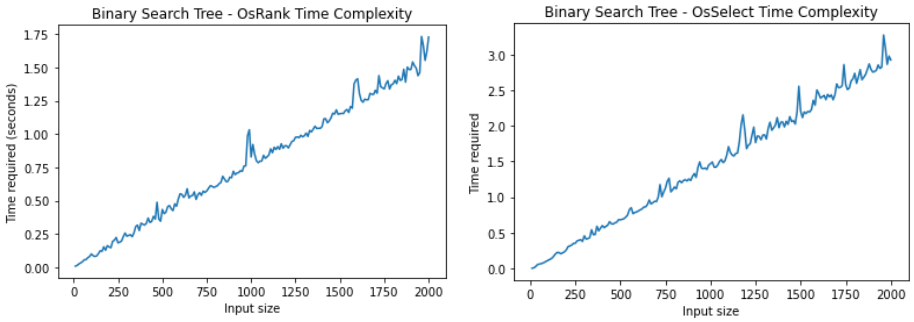
(a) Os-Rank (b) Os-Select

Figura 6: Complessità stimate sull'albero rosso nero (Struttura classica per le statistiche d'ordine dinamiche)

Dopo svariati tentativi si nota chiaramente che la complessità sarà sempre logaritmica, perciò si ha $O(logn)$ per entrambe le funzioni. Per Os-Select il grafico del logaritmo è in base minore di 2 probabilmente perchè os-select è stata implementata per scartare a priori dei nodi di cui la funzione search non riesce a trovare la chiave, restituendo quindi null.

4.2.3 Albero binario di ricerca

Il procedimento è il medesimo a sopra e l'andamento atteso è stato infatti lineare come per la lista ordinata, con una grossa differenza però nei tempi di esecuzione, come si può vedere dalla tabella a fine paragrafo. Qui non è stato necessario fare le medie.



(a) Os-Rank (b) Os-Select

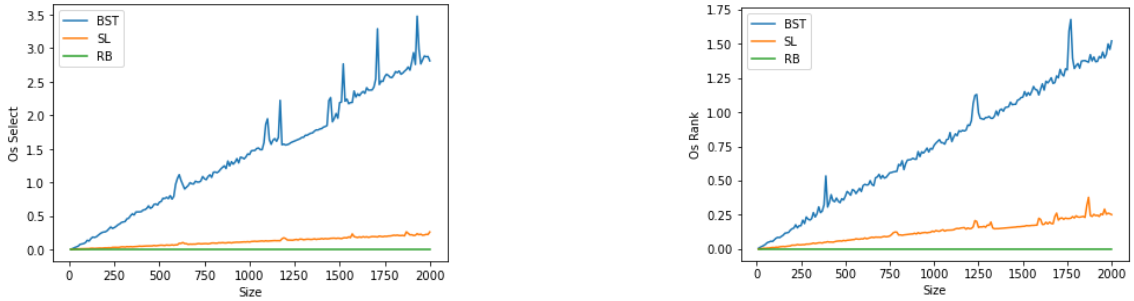
Figura 7: Complessità stimate sull'albero binario di ricerca

Abbiamo nuovamente un $O(n)$, ma con tempi decisamente maggiori a causa del dover scorrere l'albero ad ogni iterazione, operazione che viene ovviata nell'albero rosso nero, in quanto l'informazione è conservata nel nodo stesso, e nella lista ordinata.

I grafici conclusivi riportati qui di seguito aiutano a visualizzare le differenze di andamento temporale riportate nella tabella.

	Os-rank	Os-Select
SL 200 elementi	0.024	0.020
SL 2000 elementi	0.233	0.216
RN 200 elementi	0.000002	0.000001
RN 2000 elementi	0.000003	0.000002
ABR 200 elementi	0.151	0.258
ABR 2000 elementi	1.460	2.653

Figura 8: Tabella riassuntiva dei tempi di esecuzione (in secondi)



(a) OS-select

(b) OS-rank

Figura 9: Confronto finale

5 Conclusioni

A parità di tempi di inserimento e ricerca, per implementare le statistiche d'ordine dinamiche la struttura dati migliore è senza dubbio l'albero rosso nero, in quanto le sue proprietà gli impediscono di ricadere nel caso peggiore $O(n)$ e la corretta gestione dell'attributo size fa si che venga mantenuto un tempo logaritmico $O(logn)$. Se per qualsiasi motivo si decide di voler provare un'alternativa, tra albero binario di ricerca e lista ordinata si preferisce sicuramente la seconda, visti i tempi di esecuzione riportati dagli esperimenti.

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.