UNIVERSITÀ
DEGLI STUDI
FIRENZE

Università degli studi di Firenze
Dipartimento di Ingegneria Informatica

Statistiche d'ordine dinamiche

Autori:
Pennelli Lorenzo Maria

N. Matricola:
7078749

Docente corso:
Marinai Simone

Corso principale:
Algoritmi e Strutture dati

Indice

Elenco delle figure

Elenco delle tabelle

1 Introduzione

1.1 Descrizione teorica del problema

Il progetto assegnato consiste nel confrontare le funzioni di statistiche di ordine dinamiche, **OS-Select** e **OS-Rank**, su tre strutture dati. In particolare abbiamo trattato come strutture dati:

- **Lista Ordinata** (è stata implementata con i puntatori).
- **Albero Binario di Ricerca standard** (senza attributi aggiuntivi).
- **Albero Rosso Nero con l'attributo size**

I test sono stati realizzati in linguaggio Python.
La relazione che segue, presenta inizialmente in modo sintetico una trattazione teorica di ciò che sarà valutato negli esperimenti, poi seguirà la documentazione del codice relativo all'implementazione delle strutture dati e dei codici di test. Successivamente è presente l'osservazione dei risultati dei test, dove verranno valutati i risultati ottenuti con le loro aspettative teoriche.

1.2 Specifiche della piattaforma di test

La piattaforma di test sarà la stessa per ogni esercizio che vedremo. Partiamo dall'hardware del computer fondamentale da conoscere per questo esercizio:

- **CPU**: Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
- **RAM**: 8GB Dual DDR4
- **SSD**: 237GB

L'ambiente in cui è stato scritto e testato il codice è l'IDE **PyCharm 2023.3.4**.
La stesura di questo testo è avvenuta con l'utilizzo dell'editor online **Overleaf**.

2 Trattazione Teorica

2.1 Definizioni delle funzioni Select e Rank

Come spiegato nell'introduzione in questo progetto valuteremo in particolare le funzioni di **OS-Select** e di **OS-Rank**. Funzioni basilari per le Statistiche d'Ordine Dinamiche.

- **OS-Select(S, i)**: restituisce l'i-esimo elemento più piccolo in un insieme ordinato.
- **OS-Rank(S, x)**: restituisce la posizione dell'elemento x in un insieme ordinato.

Troviamo la loro utilità nel cercare o determinare la posizione in dataset di grandi dimensioni o con dimensioni sempre variabili. Grazie a loro possiamo sempre sapere il minimo, il massimo e la mediana. *(Nelle sezioni successive per semplicità OS-Select e OS-Rank verranno chiamati Select e Rank).*

2.2 Implementazione su Strutture Diverse

Abbiamo confrontato 3 diverse implementazioni di statistiche d'ordine dinamiche per vedere come variano in base alla struttura dati. Le strutture dati prese in esame sono: **Lista ordinata**, **Albero Binario di Ricerca standard**, **Albero Rosso Nero aumentato con l'attributo Size**.

Lista Ordinata (lista collegata ordinata) Questa struttura mantiene gli elementi ordinati in maniera crescente, utilizzando i puntatori tra i nodi.

- **Select(S, i)**: per accedere all'elemento in posizione i, è necessario scorrere sequenzialmente i primi i nodi.
  Complessità: $O(i)$ e nel caso peggiore $O(n)$
- **Rank(S, x)**: per contare quanti elementi sono minori o uguali a x, si deve scorrere tutta la lista confrontando ciascun nodo.
  Complessità: $O(n)$

Considerazioni: semplice da implementare, ma inefficiente per dataset di grandi dimensioni.

Albero Binario di Ricerca (BST) Struttura gerarchica in cui ogni nodo ha un figlio sinistro con valori minori e un figlio destro con valori maggiori. Tuttavia, non mantenendo informazioni aggiuntive come la dimensione del sottoalbero, le operazioni statistiche risultano complesse.

- **Select(S, i)**: senza l'attributo `size`, si deve effettuare una visita *in-order* fino a raggiungere il nodo in posizione i.
  Complessità: $O(n)$

- **Rank(S, x)**: si conta quanti nodi hanno valore minore o uguale a x, visitando potenzialmente l'intero albero.
  Complessità: $O(n)$

Considerazioni: se l'albero è sbilanciato, può degenerare in una lista, peggiorando drasticamente le prestazioni.

Albero Rosso-Nero con attributo `size` Struttura bilanciata che garantisce altezza logaritmica e include un attributo `size` in ogni nodo, rappresentante la dimensione del sottoalbero radicato in quel nodo.

- **Select(S, i)**: si utilizza l'attributo `size` per decidere se cercare nel sottoalbero sinistro, nel destro, o se il nodo corrente è quello cercato.
  Complessità: $O(\log n)$
- **Rank(S, x)**: seguendo il cammino di ricerca verso x, si sommano le dimensioni dei sottoalberi sinistri visitati e si ottiene così il rango.
  Complessità: $O(\log n)$

Considerazioni: struttura molto efficiente, ideale per operazioni dinamiche su insiemi ordinati, grazie al bilanciamento e all'uso di attributi ausiliari.

Struttura Dati	**Select**	**Rank**
Lista Ordinata	$O(i)$	$O(n)$
Albero Binario di Ricerca	$O(n)$	$O(n)$
Albero Rosso-Nero con `size`	$O(\log n)$	$O(\log n)$

Tabella 1: Complessità delle operazioni `Select` e `Rank` con diverse strutture dati

2.3 Note sull'Albero Rosso-Nero con `size`

L'Albero Rosso-Nero con l'attributo `size` è un classico esempio di struttura dati aumentata. L'aggiunta dell'attributo `size`, che rappresenta la dimensione del sottoalbero radicato in ciascun nodo, consente di eseguire efficientemente operazioni come `Select` e `Rank` in tempo logaritmico.
Tale modifica non altera la complessità delle operazioni fondamentali dell'albero (inserimento, cancellazione, ricerca), che rimangono anch'esse in $O(\log n)$. Questo rende l'albero rosso-nero aumentato particolarmente adatto al contesto dell'elaborazione di statistiche d'ordine, senza dover ideare strutture dati completamente nuove.

Limitazioni L'aggiunta dell'attributo `size` comporta un modesto aumento dello spazio occupato da ciascun nodo e introduce una leggera complessità nella gestione delle operazioni dinamiche. In particolare, durante l'inserimento e la cancellazione, è necessario aggiornare correttamente l'attributo `size` lungo il cammino di risalita, nonché durante eventuali rotazioni strutturali.
Tuttavia, questi aggiornamenti possono essere effettuati localmente in tempo costante per ogni nodo coinvolto, mantenendo la complessità complessiva delle operazioni pari a $O(\log n)$.

3 Documentazione Codice

3.1 Struttura del progetto

Il progetto è organizzato secondo una struttura ad albero, che suddivide logicamente i file in base alla loro funzione. Di seguito si riporta l'alberatura dei file e delle directory principali:

```
.
+-- input
+-- output/
|   +-- plots/
|   |    +-- LinkedList/
|   |    |   +-- random/
|   |    |   |   +-- select.png
|   |    |   |   +-- rank.png
|   |    |   +-- randomDuplicate
|   |    |   +-- sorted
|   |    |   +-- reverse
|   |    +-- BinarySearchTree
|   |    +-- confronto
|   |    +-- RedBlackTree
|   |    +-- RBTLarge
|   |    +-- table
|   +--benchmark_results.csv
+-- StruttureDati/
|   +-- Node/
|   |   +-- NodeList.py
|   |   +-- TreeNode.py
|   |   +-- NodeRedBlack.py
|   +-- ABR.py
|   +-- OrderedList.py
|   +-- ARN.py
+-- testFunctions/
|   +-- ARNValutation.py
|   +-- CreateInput.py
|   +-- BenchmarkSelectRank.py
|   +-- CreateTables.py
|   +-- PlotResults.py
+-- main.py
```

3.1.1 Directory input e output

- **Input**: Contiene i file di input utilizzati per i test sperimentali. Questi file vengono generati automaticamente da script presenti nella directory **testFunctions/CreateInput.py** che popola questa cartella di file **.joblib** che salvano le strutture dati e informazioni utili al rank. Così evitiamo di creare e ripopolare ogni volta le strutture dati.
- **Output**: Raccoglie tutti i risultati prodotti durante l'esecuzione degli esperimenti. Al suo interno si trova la cartella **plots/**, che contiene i grafici generati per ciascuna struttura dati e scenario di test. Inoltre, sono presenti i file csv dei tempi dei benchmark.

3.2 Strutture Dati

Per svolgere i nostri esperimenti ho, prima di tutto, scritto il codice delle strutture dati a cui faremo riferimento. Le classi sono **BinarySearchTree** presente nel file **ABR.py**, **TreeRedBlack** presente nel file **ARN.py**, **OrderedLinkedList** presente nel file **ListaOrdinata.py** e poi gli eventuali nodi usati, presenti negli omonimi file, **NodeList**, **NodeRedBlack**, **TreeNode**. La classe **BinarySearchTree** rappresenta la struttura dati dell'albero di binario di ricerca e "usa" la classe **TreeNode**. La classe **TreeRedBlack** raffigura un albero rosso nero e possiede la classe **NodeRedBlack**. L'ultima struttura rimasta è la **OrderedLinkedList** che rappresenta la lista ordinata, questa lista è stata implementata senza l'uso di nessuna libreria python, ma usando **NodeList** che ci permette di creare su python una lista collegata con i puntatori grazie ai suoi attributi **left** e **right** che collegano il nodo ad altri elementi della lista. Come si può notare dal Class Diagram sottostante la classe **TreeNode** estende la classe **NodeList**. Stessa cosa accade tra **NodeRedBlack** e **TreeNode**. Come si potrebbe , invece, pensare, la classe **TreeRedBlack**, nonostante l'albero rosso nero possa essere visto come una versione più bilanciata di un albero binario di ricerca, non estende la classe **BinarySearchTree**. Questa scelta è dovuta al fatto che il focus dell'esperimento sia il valutare i tempi dei **Select** e dei **Rank** e quindi ho realizzato solo i metodi necessari che si sono rilevati molto diversi tra le strutture dati.

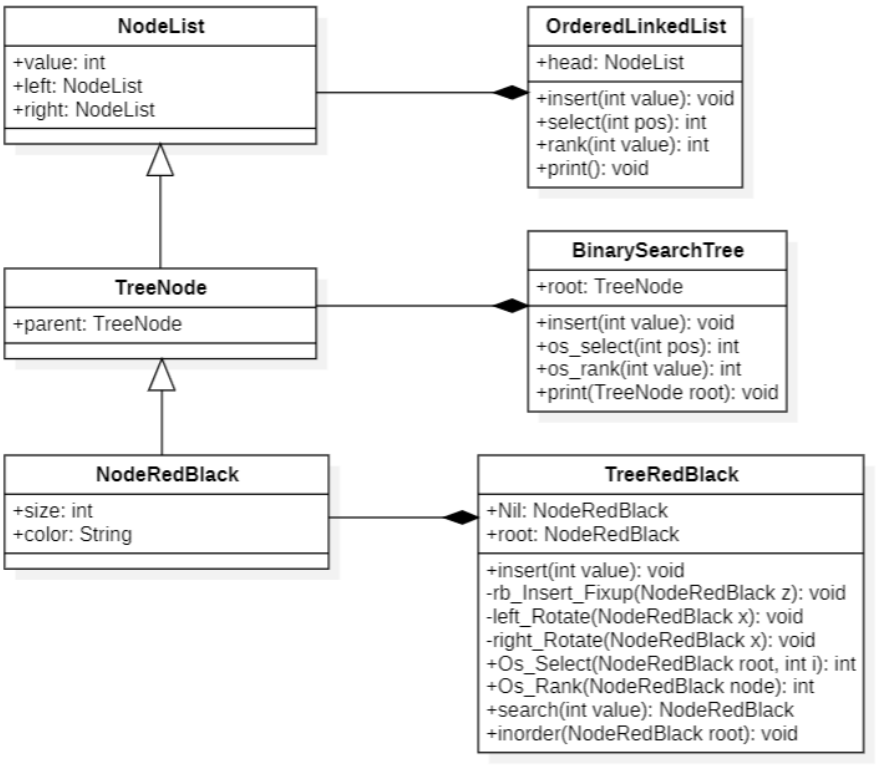


Figura 1: Diagramma di classe per le strutture dati

3.3 Funzioni di test

Per valutare in modo efficace le prestazioni delle varie strutture dati implementate, è stata predisposta una suite di script di test contenuti nella directory **testFunctions/**. Ognuno di questi script ha un compito specifico nel processo di generazione dei dati, esecuzione dei benchmark e produzione dei risultati grafici.

- **CreateInput.py**: genera automaticamente diversi dataset da utilizzare per i test, salvandoli in formato `.joblib` all'interno della directory **input/**. I dataset variano per tipo di ordinamento (random, ordinato, inverso, con duplicati) e dimensione, al fine di valutare il comportamento delle strutture in vari scenari.
- **BenchmarkSelectRank.py**: è il cuore della fase sperimentale. Questo script esegue il confronto sistematico delle strutture dati, prendendo i file dalla cartella **input/** e applicando operazioni di `Select` e `Rank` sui dataset generati. I tempi vengono registrati per ciascuna operazione, per ogni struttura dati e per ogni tipo di input in file csv.

- **PlotResults.py**: prende in input i file `.csv` generati e crea i grafici presenti nella cartella **output/plots/**. I grafici rappresentano l'andamento dei tempi in funzione della dimensione del dataset, distinguendo tra diversi tipi di ordinamento e struttura dati.
- **CreateTables.py**: genera delle tabelle riassuntive in formato `.png` che mostrano le statistiche ottenute dalle esecuzioni, organizzandole in modo che possano essere successivamente usate per un'eventuale analisi. Salvano in tabella raggruppando per operazione e tipo di ordinamento di dato i tempi confrontati di ogni struttura dati con la loro dimensione (*essendo che vengono effettuati test su tante dimensioni diverse, 30, tali tabelle non verranno usate nella loro interezza , ma verranno prese in versioni più ridotte. Tuttavia se si è interessati alle tabelle in formato originale si possono ritrovare in* ***output/plots/tables***)
- **ARNValutation.py**: contiene il codice che permette di eseguire i test specifici sull'albero Rosso-Nero. Vengono calcolati i tempi delle operazioni di **Select** e **Rank** con dataset molto più grandi, tali risultati vengono salvati in un file csv e poi vengono trasformati in un grafico. Ho dovuto creare un ulteriore file apposito perché i dataset studiati nel **BenchmarkSelectRank.py** erano troppo piccoli e quindi risolti in troppo poco tempo (solo per l'albero rosso nero) avendo così dei risultati molto soggetti al rumore generato dal computer. Inoltre, ho applicato questa separazione solo per l'albero rosso nero perché con valori troppo grandi le altre strutture dati ritardavano troppo gli esperimenti e causavano spesso crash del computer o timeout.

L'esecuzione di tutti gli script è orchestrata dal file **main.py**, che funge da punto di ingresso e permette di automatizzare l'intero processo: dalla generazione dei dati all'analisi visiva dei risultati.

3.4 Analisi delle scelte implementative

Nella progettazione delle classi e nella definizione delle loro relazioni gerarchiche, si è cercato di aderire il più possibile allo pseudocodice proposto nel testo "Introduzione agli Algoritmi e alle Strutture Dati Mc Graw Hill", al fine di mantenere coerenza con le specifiche teoriche e facilitare la corrispondenza tra teoria e implementazione. Infatti, nella classe **TreeRedBlack** è stato aggiunto un attributo Nil che rappresenta una sentinella (T.Nil), il cui unico scopo è quello di indicare la fine dell'albero e il padre della radice, senza generare possibili errori dovuti a gestioni sbagliate di None.
È stata implementata come un **NodeRedBlack** che viene instanziato alla creazione di un albero rosso nero e ha i seguenti valori dei suoi attributi:

- value= None
- color= 'Black'
- size= 0 *(così non influenza il calcolo del size)*
- parent, left, right = Nill

3.5 Descrizione dei metodi implementati

In questa parte descrivero le funzionalità di ogni metodo delle classi di cui finora abbiamo parlato.

- **OrderedLinkedList**:
  - **insert(value)**: Inserisce un nuovo nodo nella lista mantenendo l'ordine crescente. Se la lista è vuota, il nodo diventa la testa. Se il valore è minore della testa, viene inserito all'inizio. Altrimenti, viene inserito nella posizione corretta scorrendo la lista.
  - **select(position)**: Restituisce il valore del nodo che si trova nella posizione indicata (indice zero-based). Scorre la lista dalla testa fino a raggiungere la posizione desiderata.
  - **rank(value)**: Restituisce l'indice (rank) del primo nodo che ha il valore richiesto. Se il valore non è presente, restituisce `None`. È utile per capire la posizione di un elemento nella lista.
  - **print()**: Stampa su console tutti i valori presenti nella lista, in ordine, a partire dalla testa.
- **BinarySearchTree**:
  - **insert(value)**: Inserisce un nodo nella posizione corretta dell'albero binario di ricerca seguendo le regole standard (valori minori a sinistra, maggiori o uguali a destra). Aggiorna i puntatori del nodo padre.

- **os_select(position)**: Implementa la selezione dell'elemento in posizione `position` (1-based) utilizzando una visita `in-order` iterativa. Restituisce il valore del nodo che si trova in quella posizione nell'ordine crescente. Ho adottato di utilizzare la versione iterativa rispetto alla ricorsiva perché il compilatore forniva questo errore: "`RecursionError: maximum recursion depth exceeded while getting the str of an object`"
- **os_rank(value)**: Calcola il rango (posizione in ordine crescente) del nodo contenente il valore specificato. Scorre l'albero con una visita `in-order` fino a trovare il nodo con quel valore e restituisce la sua posizione. L'implementazione della visita `in-order` è stato implementato a livello iterativo sempre per il motivo specificato precedentemente.
- **print(root)**: Stampa i valori dei nodi dell'albero in ordine crescente, eseguendo una visita ricorsiva in-order a partire dalla radice.

- **TreeRedBlack**:
  - **insert(value)**: Inserisce un nuovo nodo rosso con valore `value` mantenendo aggiornati i campi `size` lungo il cammino di inserimento.
  - **rb_Insert_Fixup(z)**: Ripristina le proprietà dell'albero rosso-nero dopo un'inserzione, gestendo i tre casi classici tramite rotazioni e ricolorazioni.
  - **left_Rotate(x)**: Esegue una rotazione a sinistra intorno al nodo `x`, mantenendo aggiornato il campo `size`.
  - **right_Rotate(y)**: Esegue una rotazione a destra intorno al nodo `y`, mantenendo aggiornato il campo `size`.
  - **Os_Select(node, i)**: Restituisce il valore del nodo con rango `i` a partire dal sottoalbero radicato in `node`, sfruttando il campo `size`. È stata adottata la versione classica presente nei normali libri di testo di Algoritmi e Strutture Dati.

```python
def Os_Select(self, node, i):
    if node == self.Nil:
        return None

    rank = node.left.size + 1
    if i == rank:
        return node.value
    elif i < rank:
        return self.Os_Select(node.left, i)
    else:
        return self.Os_Select(node.right, i - rank)
```

Figura 2: Funzione Select dell'Albero Rosso Nero

- **Os_Rank(node)**: Calcola il rango del nodo `node` (cioè il numero di elementi minori di esso nell'albero) risalendo fino alla radice. È stata adottata anche qui la versione classica presente nei normali libri di testo di Algoritmi e Strutture Dati.

```
def Os_Rank(self, node):
    if node == self.Nil:
        return 0

    rank = node.left.size + 1
    while node != self.root:
        if node == node.parent.right:
            rank += node.parent.left.size + 1
        node = node.parent
    return rank
```

Figura 3: Funzione Rank dell'Albero Rosso Nero

- **inorder(node)**: Stampa in ordine crescente i valori dei nodi nel sottoalbero radicato in `node`, insieme al rispettivo `size`.
- **search(value)**: Restituisce il nodo con valore `value` se presente, altrimenti `None`.

- **CreateInput**:
  - **creation_data_structure()**:
    ha il compito di generare dataset di input secondo quattro configurazioni: `random`, `sorted`, `reverse` e `randomDuplicate`. Per ciascuna di queste configurazioni, i dati vengono generati per diverse dimensioni predefinite, da 10 fino a 1400 elementi (limite massimo che la libreria joblib può gestire per salvare le strutture dati senza eccedere il proprio limite personale di ricorsione). Ogni insieme di dati viene utilizzato per popolare le tre diverse strutture dati. Dopo l'inserimento, viene eseguita l'operazione `select` su ciascuna struttura per ricavare un valore centrale (la posizione mediana, calcolata come `size // 2`). Successivamente, il valore centrale selezionato e le tre strutture dati vengono salvati su file in formato `.joblib`.

- **BenchmarkSelectRank**:
  - **benchmark_from_saved_structures(folder, output_csv, num_trials)**:
    La funzione esegue un benchmark delle operazioni `select` e `rank` su tre strutture dati: **OrderedLinkedList**, **BinarySearchTree** e **RedBlackTree**, utilizzando i dati precedentemente salvati su file. Vengono caricati i file serializzati in formato `joblib` da una cartella specificata (default: `"input"`) e si misura il tempo medio

(in millisecondi) impiegato per eseguire le due operazioni principali su ciascuna struttura. La misurazione è ripetuta per un numero di prove specificato da `num_trials` (default: 5), per ridurre la variabilità. I risultati, contenenti tipo di dataset, dimensione, e tempi medi delle operazioni, vengono salvati in un file CSV (default: `"output/benchmark_results.csv"`) e restituiti anche come `DataFrame` Pandas per un'eventuale analisi successiva. Sono gestiti eventuali errori durante il caricamento dei file o l'esecuzione delle operazioni.

```
def avg_measure(func):
    try:
        times = [timeit.timeit(func, number=1) * 1000 for _ in range(num_trials)]
        return np.mean(times)
    except Exception as e:
        print(f" Errore durante il timing: {e}")
        return float('nan')
```

Figura 4: Funzione che misura i tempi di esecuzione

- **PlotResults**:
  - **plot_benchmarks(csv_path)**
    Questa funzione si occupa di generare e salvare dei grafici a partire dai risultati di benchmark contenuti in un file CSV (default: `"output/benchmark_results.csv"`). I dati vengono caricati in un `DataFrame` Pandas, e vengono create due tipologie principali di grafici: un confronto diretto tra tutte le strutture dati per ciascun tipo di dataset e operazione, e una visualizzazione dettagliata delle performance di ogni singola struttura per ciascun tipo di dataset.
- **CreateTables**:
  - **generate_operation_tables_from_csv(csv_path, output_folder)**
    Questa funzione prende in input il percorso a un file CSV contenente i risultati dei benchmark e genera tabelle comparative in formato immagine (`.png`) per ogni combinazione di operazione (`Select`, `Rank`) e tipo di dataset (`random`, `sorted`, `reverse`, `randomDuplicate`). Ogni tabella viene formattata per mostrare i tempi medi in millisecondi con precisione a 4 cifre significative e infine salvata come immagine nella cartella di output predefinita
    (`output/plots/table/<operation>/<datatype>.png`)
- **ARNValutation**
  - **benchmark_rbt_select_rank(**
    trials=1000, output_dir="output/plots/RBTLarge")
    Questa funzione esegue un benchmark dettagliato delle operazioni `Select` e `Rank` su alberi Red-Black (ARN) per varie dimensioni di

dati, da 100 fino a 1 milione di nodi. Per ogni dimensione della lista predefinita `sizes`, viene generato un dataset di numeri interi unici in ordine casuale. Successivamente, viene creato un albero Red-Black vuoto e tutti i valori del dataset vengono inseriti nell'albero. Viene quindi selezionato un valore centrale, calcolato come `i_select = size // 2`, che viene utilizzato sia per la funzione `Select` (che esegue la ricerca per rango) sia per la funzione `Rank` (che calcola il rango di un nodo). Le funzioni `Os_Select` e `Os_Rank` vengono eseguite ripetutamente per un numero di volte definito da `trials`, in modo da misurarne il tempo medio di esecuzione espresso in millisecondi. I risultati dei tempi medi per ogni dimensione sono poi salvati in un DataFrame Pandas e esportati in un file CSV. Infine, la funzione genera e salva due grafici a linea, uno relativo ai tempi della funzione `Select` e l'altro ai tempi della funzione `Rank`.

4 Descrizione degli esperimenti condotti e analisi dei risultati sperimentali

4.1 Dati utilizzati

L'esperimento che ho svolto si divide prima di tutto nel tipo di dati generati da inserire nelle strutture dati. L'esperimento sarà diviso in 4 parti:

- dati **randomici** senza duplicati.
- dati **ordinati** senza duplicati.
- dati **ordinati al contrario** senza duplicati.
- dati **randomici con duplicati**

Con questa divisione voglio vedere se i tempi del **Select** e **Rank** vengano influenzati dalla disposizione dei dati e se ci sono eventuali cambiamenti se siamo nel caso migliore o nel caso peggiore della struttura dati (*in particolare la lista ordinata e l'albero binario di ricerca*).
Il range delle dimensioni delle strutture dati vanno da 10 elementi fino a 1400 osservare al meglio l'andamento con l'aumentare della dimensione.

```
data_configs = {
    'random': lambda n: random.sample(range(1, n + 1), n),
    'sorted': lambda n: sorted(random.sample(range(1, n + 1), n)),
    'reverse': lambda n: sorted(random.sample(range(1, n + 1), n), reverse=True),
    'randomDuplicate': lambda n: [random.randint( a: 1, n) for _ in range(n)]
}
```

Figura 5: Frammento di codice che mostra la generazione dei dati nelle 4 casistiche in base alla dimensione

4.2 Risultati sperimentali

4.2.1 Dati randomici senza duplicati

Questo caso rappresenta il caso "**medio**" di tutte e tre le strutture dati. Mostriamo qui di seguito il grafico con tutte le dimensioni

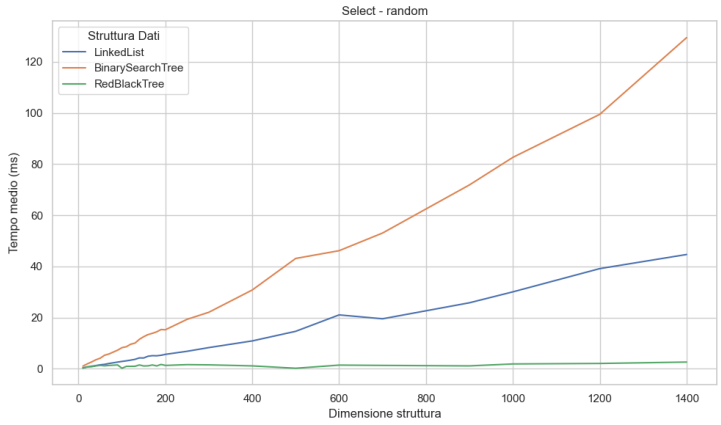


Figura 6: Grafico del Select con dati random

Size	Albero Binario	Lista Ordinata	Albero Rosso Nero
10	1.054	0.4283	0.23
30	2.681	1.015	0.8211
180	14.47	5.089	1.144
1400	129.5	44.7	2.643

Tabella 2: Tempi medi di esecuzione (ms) delle operazioni `Select` con dati di input random

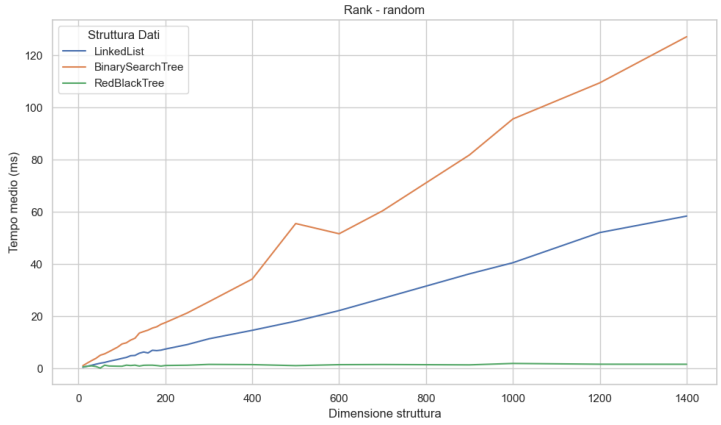
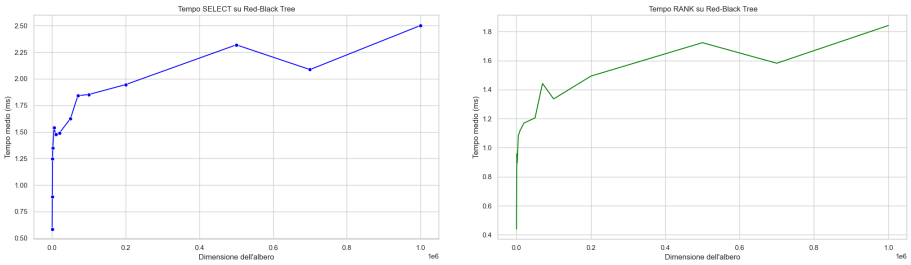


Figura 7: Grafico del Rank con dati random

Size	Albero Binario	Lista Ordinata	Albero Rosso Nero
10	1.15	0.535	0.6903
30	3.075	1.295	1.065
180	16.03	6.914	1.193
1400	127.1	58.1	1.688

Tabella 3: Tempi medi di esecuzione (ms) delle operazioni `Rank` con dati di input random



(a) Grafico del Select dell'Albero Rosso Nero

(b) Grafico del Rank dell'Albero Rosso Nero

Figura 8: Tempi medi delle operazioni **Select** e **Rank** su Red-Black Tree

Notiamo dalle figure 6 e 7 che l'**albero binario di ricerca** e la **lista ordinata** hanno tempi maggiori rispetto all'**albero rosso nero**. Inoltre, queste due strutture dati, già con pochi elementi, mostrano un andamento lineare $O(n)$. La **lista ordinata** risulta leggermente più efficiente rispetto all'**albero binario di ricerca** nella funzione `Select`, soprattutto in presenza di strutture di

piccole e medie dimensioni. Questo comportamento è dovuto principalmente al fatto che l'inserimento e la ricerca sequenziale in una lista collegata hanno un andamento prevedibile e lineare, mentre l'implementazione dell'albero binario di ricerca, non essendo bilanciato, può degradare in una struttura sbilanciata, con prestazioni significativamente peggiori.
L'**albero rosso nero** è talmente più efficiente che è stato valutato anche singolarmente con altri valori, anche per evitare eventuali rumori del computer, (figura 8.a e figura 8.b) mostrando un andamento simil $O(logn)$. Inoltre, il diverso ordine dei dati di input non lo influenza, grazie al suo riordinamento interno (**rb_Insert_Fixup**) e quindi per ogni casistica di input si farà sempre esempio ai seguenti grafi. *(per maggiori informazioni sezione 4.3).*

4.2.2 dati ordinati senza duplicati

Abbiamo preso in considerazione questo caso perché rappresenta il caso **peggiore** per l'albero binario di ricerca, dato che comporta uno sbilanciamento di quest'ultimo in una lista ordinata, e della lista ordinata. Mostriamo qui di seguito il grafico:

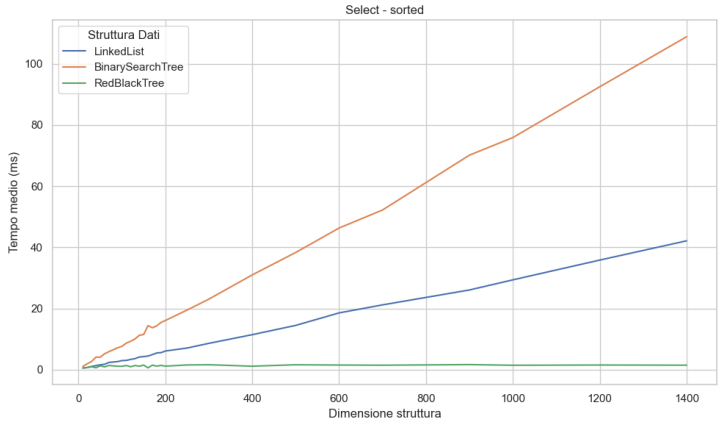


Figura 9: Grafico del Select con dati ordinati

Size	Albero Binario	Lista Ordinata	Albero Rosso Nero
10	1.095	0.5392	0.6353
30	2.692	1.093	0.962
180	14.34	5.49	1.186
1400	108.8	42.12	1.494

Tabella 4: Tempi medi di esecuzione (ms) delle operazioni `Select` con dati di input ordinati

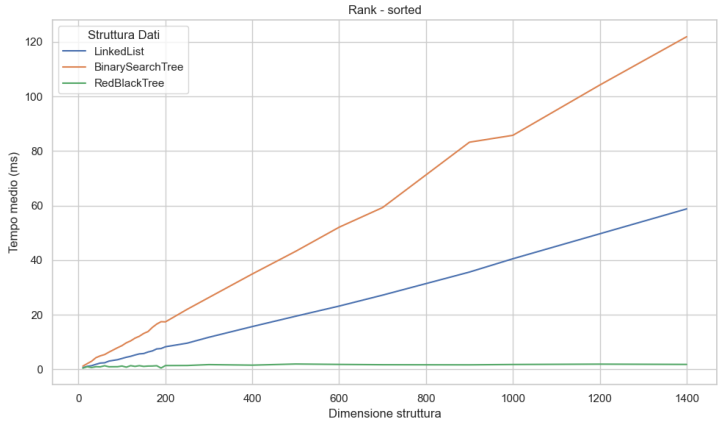


Figura 10: Grafico del Rank con dati ordinati

Size	Albero Binario	Lista Ordinata	Albero Rosso Nero
10	1.174	0.616	0.44
30	2.957	1.321	0.654
180	16.57	7.473	1.311
1400	121.9	58.78	1.796

Tabella 5: Tempi medi di esecuzione (ms) delle operazioni `Rank` con dati di input ordinati

4.2.3 dati ordinati al contrario

Questo caso è stato preso in considerazione perché crea uno sbilanciamento dell'albero binario di ricerca, ma è il caso **migiore** della lista ordinata.

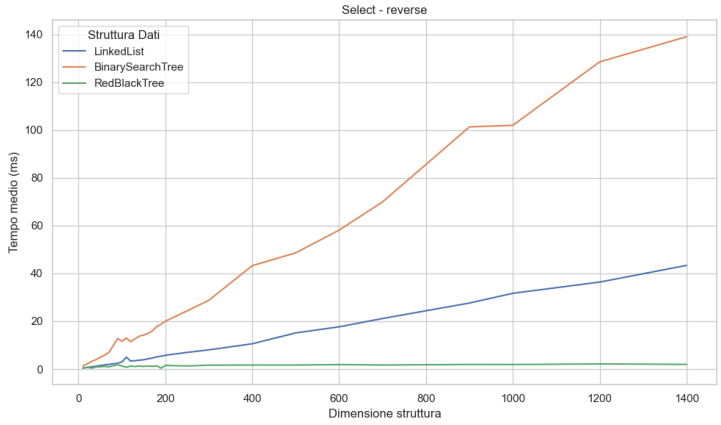


Figura 11: Grafico Select con dati ordinati al contrario

Size	Albero Binario	Lista Ordinata	Albero Rosso Nero
10	1.309	0.423	0.411
30	3.166	0.956	0.5
180	17.79	5.111	1.341
1400	139.1	43.38	1.969

Tabella 6: Tempi medi di esecuzione (ms) delle operazioni `Select` con dati di input ordinati al contrario

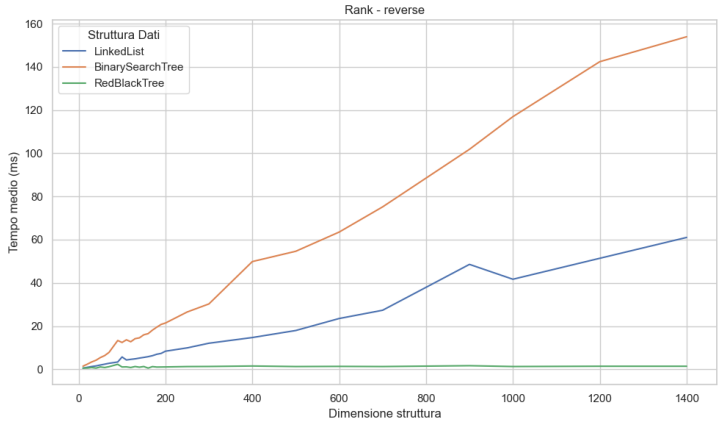


Figura 12: Grafico Rank con dati ordinati al contrario

Size	Albero Binario	Lista Ordinata	Albero Rosso Nero
10	1.432	0.5313	0.5364
30	3.382	1.265	0.797
180	19.49	6.946	1.019
1400	154	61.03	1.393

Tabella 7: Tempi medi di esecuzione (ms) delle operazioni `Rank` con dati di input ordinati al contrario

4.2.4 dati random con duplicati

Questo caso è stato scelto per vedere se la presenza di duplicati influenzasse in modo positivo o negativo le funzioni di **Select** e **Rank** nelle diverse strutture dati. Qui mostreremo i risultati.

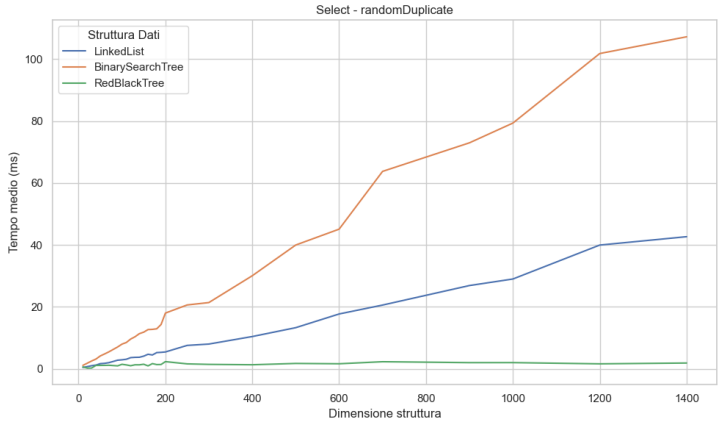


Figura 13: Grafico Select con dati random duplicati

Size	Albero Binario	Lista Ordinata	Albero Rosso Nero
10	1.118	0.442	0.6264
30	2.523	1.004	0.89
180	12.93	5.241	1.377
1400	107.2	42.68	1.876

Tabella 8: Tempi medi di esecuzione (ms) delle operazioni `Select` con dati di input random duplicabili

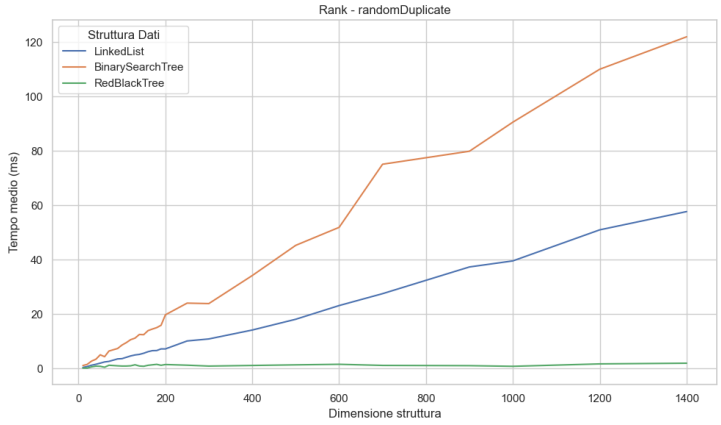


Figura 14: Grafico Rank con dati random duplicati

Size	Albero Binario	Lista Ordinata	Albero Rosso Nero
10	1.155	0.4584	0.223
30	2.791	1.28	0.678
180	15.09	6.67	1.595
1400	122	57.73	2.013

Tabella 9: Tempi medi di esecuzione (ms) delle operazioni `Rank` con dati di input random duplicabili

4.3 Approfondimenti risultati

All'interno della relazione per questione di rilevanza e di estetica non sono stati inseriti tutti i grafici ottenuti dai test, come ad esempio tutti i risultati delle tabelle o l'andamento di una singola struttura dati.
Per completezza inserisco dei link dove è possibile osservarli:

- **Tabelle complete**
- **Albero Binario di Ricerca**
- **Lista Ordinata**
- **Albero Rosso Nero**

4.4 Tesi e sintesi finale

Osservando i risultati ottenuti dai test e proposti nella sezione 4.2 possiamo dare le seguenti osservazioni:

- i metodi di **select** e di **rank** nell'**albero binario di ricerca** e **lista ordinata** hanno complessità $O(n)$, mentre l'**albero rosso nero aumentato** dimostra una complessità $O(logn)$.
- la complessità dei metodi **select** e di **rank** non vengono influenzati se siamo nel caso migliore o peggiore di una struttura dati. Infatti, anche se l'**albero binario di ricerca** risulti sbilanciato esso ci mette sempre $O(n)$. Notiamo che la stessa cosa accade anche per la **lista ordinata**. L'ordine degli input e la loro distribuzione non influenza queste operazioni.
- L'**albero rosso-nero**, grazie al bilanciamento automatico e alla presenza dell'attributo **size**, risulta sistematicamente più efficiente rispetto alle altre strutture. Tuttavia, la sua elevata efficienza rende più difficile osservare chiaramente l'andamento asintotico, soprattutto in presenza di dataset di dimensioni ridotte.
- Sebbene l'albero rosso-nero comporti un lieve overhead in termini di memoria e complessità implementativa, esso garantisce una scalabilità molto più stabile e prevedibile con la crescita del dataset.

In definitiva, l'analisi condotta ha confermato le previsioni teoriche, dimostrando come le prestazioni delle statistiche d'ordine dipendano fortemente dalla struttura dati adottata.

Riferimenti bibliografici

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein Introduzione agli algoritmi e strutture dati Terza edizione, McGraw Hill.