UNIVERSITÀ
DEGLI STUDI
FIRENZE

UNIVERSITÀ DEGLI STUDI DI FIRENZE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Relazione di Algoritmi e Strutture Dati

Autore:
Leonardo Viti

Docente
docente

Indice

1 Introduzione

Questa relazione analizza le prestazioni di diverse strutture dati nell'implementazione delle statistiche d'ordine dinamiche, in particolare le operazioni di `select(k)` e `rank(x)`. Verranno descritte le strutture dati usate, esposte le scelte progettuali fatte e alcuni aspetti implementativi. Di seguito verranno messe a confronto con opportuni test le prestazioni nelle operazioni di selezione dell' i-esimo elemento più piccolo e di calcolo del rango di un elemento. Lo scopo di questa analisi è confrontare tre strutture dati differenti:

- Lista concatenata ordinata (`LinkedList`)
- Albero binario di ricerca standard senza attributo dimensione (`BSTree`)
- Albero binario di ricerca auto bilanciante (AVL) con attributo dimensione (`SBSTree`)

Verranno presentati i risultati sperimentali che evidenziano come le diverse implementazioni si comportano al variare delle dimensioni e tipologia di dati in ingresso.

1.1 Esercizio

Esercizio assegnato per l'esame di Laboratorio di Algoritmi e Strutture Dati:

> Vogliamo confrontare varie implementazioni di statistiche d'ordine dinamiche:
>
> 1. Con lista ordinata
> 2. Con ABR senza attributo *size*
> 3. Come visto a lezione (AVL) con attributo *size*
>
> Nota: La lista deve essere implementata considerando strutture collegate con puntatori e non la struttura dati lista di Python.

2 Descrizione del problema e implementazione

In questo progetto verranno confrontate le prestazioni delle statistiche di ordine dinamiche in 3 diverse strutture dati. Verranno descritte le strutture dati utilizzate ed esposte le scelte progettuali e implementative. Successivamente verranno confrontate con opportuni esperimenti le prestazioni nelle operazioni select(k) e rank(x) per ogni struttura dati, il che comporta queste due operazioni fondamentali: select(k) e rank(x) devono essere implementate per ogni struttura dati scelta a seconda delle loro caratteristiche per permettere rispettivamente di selezionare l'elemento di rango k e di calcolare il rango di un elemento x.

2.1 Statistiche d'Ordine Dinamiche

Le statistiche d'ordine sono misure che descrivono la distribuzione degli elementi in un insieme ordinato. Nel contesto delle strutture dati dinamiche, ci concentriamo su due operazioni fondamentali:

- **select(k)**: restituisce l'elemento di rango k (il k-esimo elemento più piccolo) nell'insieme.
- **rank(x)**: restituisce il numero di elementi strettamente minori di x nell'insieme.

Queste operazioni consentono di estrarre informazioni sulla posizione relativa degli elementi in una collezione ordinata dinamica, dove elementi possono essere aggiunti o rimossi nel tempo. Sono state implementate tre diverse strutture dati per supportare queste operazioni: una lista concatenata ordinata, un albero binario di ricerca standard e un albero binario di ricerca auto bilanciato (AVL) con attributo dimensione.

2.2 Implementazione

2.2.1 Lista Concatenata Ordinata

La lista concatenata ordinata è una struttura dati dinamica i cui elementi sono mantenuti in ordine crescente; è implementata con nodi che contengono un valore e un puntatore al nodo successivo.

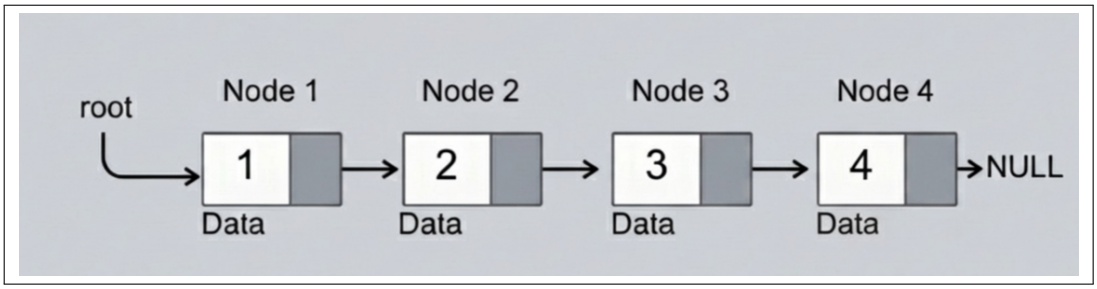


Figura 1: struttura lista concatenata

2.2.2 Implementazione lista concatenata ordinata

È stato scelto di implementare la lista come singolarmente concatenata 1per semplicità, in quanto le operazioni richieste non necessitano di accesso bidirezionale. Il puntatore `root` punta al primo nodo della lista. L'inserimento di un nuovo elemento avviene scorrendo la lista fino a trovare la posizione corretta per mantenere l'ordine crescente, quindi si aggiorna il puntatore del nodo precedente per puntare al nodo appena inserito e il puntatore del nuovo nodo per puntare al nodo successivo. La cancellazione di un elemento avviene scorrendo la lista per trovare il nodo da rimuovere, quindi si aggiorna il puntatore del nodo precedente per saltare il nodo da rimuovere.

2.2.3 Albero Binario di Ricerca Standard

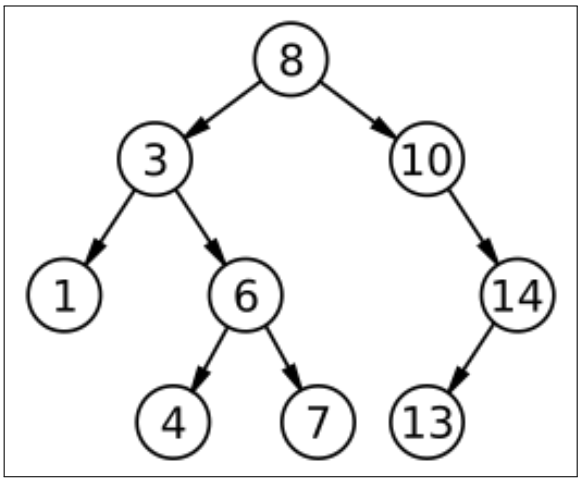


Figura 2: struttura albero binario di ricerca

L'albero binario di ricerca (BST) 2 è una struttura dati organizzata in un albero binario: un grafo non orientato, connesso e aciclico; quindi ogni nodo può avere massimo due figli con il nodo iniziale detto radice dal quale si diramano tutti gli altri nodi e che ha il puntatore al padre nullo. Ogni nodo è un oggetto e ha 4 campi: valore(la chiave del nodo quindi il dato), puntatore al figlio sinistro, puntatore al figlio destro, puntatore al padre. Ogni nodo segue la proprietà fondamentale degli alberi binari di ricerca.
La propietà fondamentale di un albero binario di ricerca è che per ogni nodo, tutti i valori nel sottoalbero sinistro sono minori del valore del nodo, e tutti i valori nel sottoalbero destro sono maggiori.

Proprietà fondamentale

Proprietà fondamentale Sia x un nodo in un ABR.
Se y è nel sottoalbero sinistro di x, allora y.key x.key
Se y è nel sottoalbero destro di x, allora y.key x.key

L'inserimento di un nuovo elemento avviene confrontando il valore da inserire con i valori dei nodi, scendendo a sinistra o a destra a seconda del confronto, fino a trovare una posizione vuota dove inserire il nuovo nodo. La cancellazione di un elemento segue le regole standard per la rimozione in un albero binario di ricerca, gestendo i casi di nodi con zero, uno o due figli.

2.2.4 Albero Binario di Ricerca auto bilanciato con Attributo Dimensione

La classe `SBSTree` è implementata basandosi su una struttura **AVL (Albero Autobilanciato)**. Questa implementazione combina due proprietà fondamentali:

1. **Bilanciamento AVL**: L'albero mantiene un fattore di bilanciamento tra i sottoalberi sinistro e destro di ogni nodo, garantendo che l'altezza rimanga sempre $h = O(\log n)$, indipendentemente dall'ordine di inserimento.

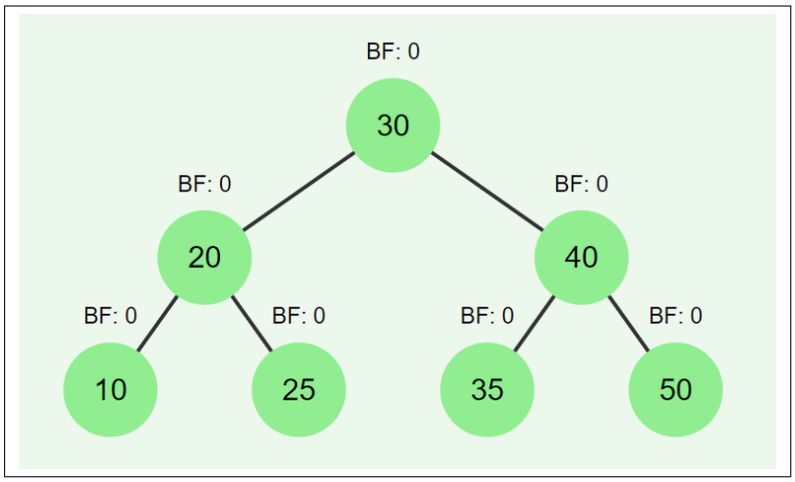


Figura 3: struttura albero AVL

Oltre all' avl standard 3, ogni nodo dell'albero contiene un attributo aggiuntivo:

2. **Attributo size aggiornato**: Ogni nodo mantiene un campo `size` che tiene traccia del numero di nodi nel sottoalbero radicato in quel nodo. Questo attributo è il nucleo dell'ottimizzazione e viene **mantenuto aggiornato durante le rotazioni AVL** che avvengono per preservare il bilanciamento.

Proprietà fondamentale AVL

Sia x un nodo in un albero AVL vale:
$\left|\text{height}\left(x_{\text{sin}}\right) - \text{height}\left(x_{\text{des}}\right)\right| \leq 1, \Rightarrow \quad h = O(\log n),$
quindi inserimento, cancellazione e ricerca sono $O(\log n)$.

Impatto sulla Complessità: La combinazione di:

- Altezza logaritmica garantita dal bilanciamento AVL: $h = O(\log n)$
- Attributo `size` accessible in $O(1)$ ad ogni nodo

Questa implementazione rappresenta l'ottimale teorico per le operazioni di statistiche d'ordine su strutture dati dinamiche.

2.3 Complessità Teorica

Tabella delle prestazioni teoriche 1 attese per select e rank nelle tre strutture dati implementate. La tabella riporta la funzione di costo di select e rank per le strutture e una breve spiegazione delle ragioni dietro queste complessità nei diversi casi.

Struttura	Operazioni	Migliore	Medio	Peggiore	Note
LinkedList	select(k)	$O(1)$	$O(n)$	$O(n)$	Accesso sequenziale dal nodo iniziale
	rank(x)	$O(1)$	$O(n)$	$O(n)$	Conteggio lineare elementi $< x$
BSTree	select(k)	$O(n)$	$O(n)$	$O(n)$	Visita in-order: $O(n)$ indip. dall'altezza
	rank(x)	$O(n)$	$O(n)$	$O(n)$	Visita sottoalberi: $O(n)$ indip. dall'altezza
SBSTree	select(k)	$O(\log n)$	$O(\log n)$	$O(\log n)$	Altezza bilanciata $h = O(\log n)$
	rank(x)	$O(\log n)$	$O(\log n)$	$O(\log n)$	Altezza bilanciata $h = O(\log n)$

Tabella 1: Complessità teorica di select e rank nelle tre strutture dati.

2.3.1 Note sulla Tabella

1. Lista Concatenata (Ordinata)

- **select(i) $\rightarrow$ T(n,i)=O(i) Spiegazione**:
  Il costo è $O(i)$ perché, per trovare l'i-esimo elemento, l'algoritmo deve partire dalla testa e scorrere i nodi. Non c'è modo di "saltare" al centro.
- **rank(x) $\rightarrow$ T(n,k)=O(k) Spiegazione**:
  Il costo è $O(k)$, dove k è il rango effettivo di x (quanti elementi sono $< x$). L'algoritmo deve scorrere la lista e contare k elementi.

2. BSTree (senza attributo size)

Complessità: $O(n)$ sia nel caso medio che nel caso peggiore

- **select(k)**: Implementata con visita **in-order iterativa**, che tocca ogni nodo fino a raggiungere la k-esima posizione. Nel caso medio ($k \approx n/2$) e peggiore ($k = n$) costa $O(n)$.
- **rank(x)**: Implementata scendendo l'albero e calcolando ricorsivamente la dimensione dei sottoalberi sinistri.

3. SBSTree (AVL + attributo size)

Complessità: $O(h) = O(\log n)$
Ragione fondamentale: La presenza dell'attributo `size` aggiornato + il bilanciamento AVL permette un algoritmo che **non visita sottoalberi interni**.

- **select(k)** e **rank(x)**: Scendono dalla radice verso il nodo obbiettivo seguendo un singolo percorso. Ad ogni nodo, in tempo $O(1)$, usano il campo `size` del figlio sinistro per confrontare k (o contare elementi $< x$) e decidere se andare a sinistra, a destra, o fermarsi. Nessun sottoalbero viene visitato completamente: solo i nodi sul cammino dalla radice al target.
- **Costo**: Proporzionale all'altezza dell'albero: $O(h)$.
- **Altezza garantita dal bilanciamento AVL**: $h = O(\log n)$ in tutti i casi (migliore, medio, peggiore).

Conclusione: Con il bilanciamento AVL, la complessità finale è $O(\log n)$, rappresentando l'ottimo teorico per le statistiche d'ordine dinamiche.
Si vuole mostrare anche che non è il bilanciamento a garantire l'efficienza, ma l'attributo size aggiornato.

3 Documentazione

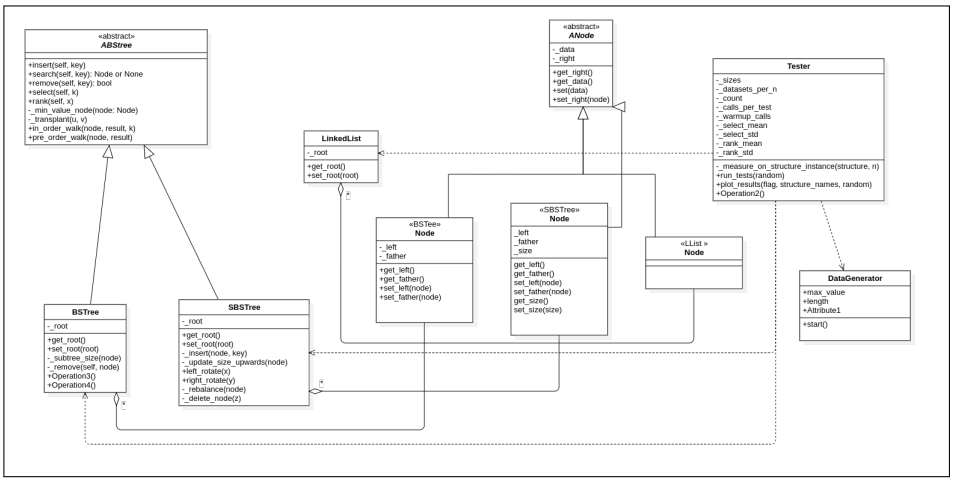


Figura 4: UML delle classi principali del progetto

3.1 Struttura del Progetto

Verranno descritte le classi principali del progetto 4 e le loro funzionalità. Tutte le implementazioni di aggiunta di un nodo per le strutture dati ricevono in ingresso solo la chiave e creano il nodo a seconda della struttura.

3.2 LinkedList

La classe `LinkedList` implementa una lista concatenata ordinata. Gli elementi sono mantenuti in ordine crescente durante l'inserimento. Metodi principali della linked list oltre a select e rank:
`insert(x)`: inserisce l'elemento x nella lista mantenendo l'ordine e modificando i puntatori del nodo da inserire e del nodo precedente;
`delete(x)`: rimuove l'elemento x dalla lista modificando i puntatori del nodo precedente e del nodo successivo;
`search(x)`: cerca l'elemento x nella lista e restituisce il nodo se trovato, altrimenti None.

```
def select(self, k):
    current = self._root
    count = 1
    while current:
        if count == k:
            return current.data
        current = current.next
        count += 1
    return None  # k troppo grande

def rank(self, x):
    current = self._root
    count = 0
    while current and current.data < x:
        count += 1
        current = current.next
    return count
```

Listing 1: Implementazione di select() e rank() in LinkedList

In questa implementazione, `select(k)` scorre linearmente la lista fino a raggiungere l'elemento di posizione k, mentre `rank(x)` conta gli elementi minori di x.

3.3 BSTree

La classe `BSTree` implementa un albero binario di ricerca standard, senza attributo dimensione nei nodi.

`insert(x)`: inserisce l'elemento x nell'albero mantenendo la proprietà di ricerca binaria;
`subtree_size(node)`: calcola la dimensione del sottoalbero avente come radice un dato nodo;
`delete(x)`: rimuove l'elemento x dall'albero mantenendo la proprietà di ricerca binaria usando i metodi standard di cancellazione in un BST;
`search(x)`: cerca l'elemento x nell'albero e restituisce il nodo se trovato, altrimenti None;
`transplant`: sostituisce un sottoalbero con un altro, utilizzato nella cancellazione dei nodi, ereditato dalla classe astratta.

```
def select(self, k):
    if k is None or k <= 0 or self._root is None:
        return None

    stack = []
    current = self._root
    count = 0

    while stack or current:
        while current:
            stack.append(current)
            current = current.get_left()
        current = stack.pop()
        count += 1
        if count == k:
            return current
        current = current.get_right()

    return None

def rank(self, x):
    current = self._root
    rank = 0
    while current:
        if x > current.get_data():
            left_size = self._subtree_size(current.get_left())
            rank += 1 + left_size
            current = current.get_right()
        else:
            current = current.get_left()
    return rank

def _subtree_size(self, node):
    if node is None:
        return 0
    return 1 + self._subtree_size(node.get_left()) +
    self._subtree_size(node.get_right())
```

Listing 2: Implementazione di select() e rank() in BSTree

In questa implementazione, `select(k)` effettua una visita in-order iterativa con conteggio degli elementi, mentre `rank(x)` calcola ricorsivamente la dimensione dei sottoalberi contenenti nodi con chiave minore di x.

3.4 SBSTree

La classe `SBSTree` implementa un albero binario di ricerca dove ogni nodo mantiene la dimensione del suo sottoalbero. `insert(x)`: inserisce l'elemento x nell'albero mantenendo la proprietà di ricerca binaria e aggiornando l'attributo dimensione dei nodi lungo il percorso di inserimento;
`delete(x)`: rimuove l'elemento x dall'albero mantenendo la proprietà di ricerca binaria e aggiornando l'attributo dimensione dei nodi lungo il percorso di cancellazione;
`search(x)`: cerca l'elemento x nell'albero e restituisce il nodo se trovato, altrimenti None;
`update_size_upwards`: aggiorna l'attributo dimensione dei nodi lungo il percorso dalla radice al nodo specificato;
`transplant)`: sostituisce un sottoalbero con un altro, utilizzato nella cancellazione dei nodi, ereditato dalla classe astratta;

`rotate_left` e `rotate_right`: eseguono rotazioni per mantenere l'equilibrio AVL, aggiornando anche l'attributo dimensione dei nodi coinvolti;
`_rebalance`: bilancia l'albero dopo inserimenti o cancellazioni, utilizzando le rotazioni AVL.
Le varie funzioni aggiornano correttamente l'attributo size dei nodi coinvolti.

```
def select(self, k):
    return self._select(self._root, k)

def _select(self, node, k):
    if node is None:
        return None
    left_size = node.get_left().get_size() if node.get_left() else 0
    if k == left_size + 1:
        return node
    elif k <= left_size:
        return self._select(node.get_left(), k)
    else:
        return self._select(node.get_right(), k - left_size - 1)

def rank(self, x):
    return self._rank(self._root, x)

def _rank(self, node, x):
    if node is None:
        return 0
    if x < node.get_data():
        return self._rank(node.get_left(), x)
    elif x > node.get_data():
        left_size = node.get_left().get_size() if node.get_left() else 0
        return 1 + left_size + self._rank(node.get_right(), x)
    else:
        left_size = node.get_left().get_size() if node.get_left() else 0
        return left_size
```

Listing 3: Implementazione di select() e rank() in SBSTree

In questa implementazione, sia `select(k)` che `rank(x)` usano la proprietà `size` di ogni nodo per determinare rapidamente la dimensione dei sotto alberi, consentendo operazioni più efficienti.

3.5 DataGenerator

La generazione dei dati di test è affidata alla classe `DataGenerator`. Il suo scopo è : fornire insiemi di numeri interi non duplicati con caratteristiche controllate per mettere alla prova le strutture dati in scenari diversi. Quando istanziata si specificano i parametri principali:
lunghezza dell'array, valore massimo, tipo di ordinamento, e il costruttore verifica la coerenza della configurazione sollevando `ValueError` quando i parametri sono in conflitto (per esempio quando si richiede l'unicità di valori ma `length` supera `max_value`).
Per comodità sperimentale la classe offre due modalità d'uso: dati casuali (`'random'`), dati già ordinati in senso crescente (`'sorted'`).
L'implementazione sfrutta `numpy` per la generazione efficiente e il mescolamento degli array.
Il metodo `start()` restituisce l'array pronto all'uso: nel caso `'random'` un vettore di interi unici mescolati, nel caso `'sorted'` una sequenza ordinata in senso crescente. il tester si occupa di usare l'array per popolare le strutture e avviare le misure.

Esempio illustrativo

```
import numpy as np

class DataGenerator:
    def __init__(self, length, max_value=None, data_type='random'):
        # validazione parametri e inizializzazione
        self.length = length
        self.max_value = max_value or length
        self.data_type = data_type

```

```
    def start(self):
        if self.data_type == 'random':
            arr = self.rng.choice(np.arange(self.max_value),
    size=self.length, replace=False)
            self.rng.shuffle(arr)
            return arr
        elif self.data_type == 'sorted':
            return np.arange(self.length)
        else:
            raise ValueError('data_type non supportato')
```

Listing 4: Esempio semplificato di DataGenerator

3.6 Funzioni di test

La classe di misurazione è organizzata per ottenere stime affidabili delle prestazioni mediando molteplici misure su insiemi di dati indipendenti. In pratica, per ogni dimensione di interesse si generano più insiemi di dati (sia casuali che ordinati a seconda dell' esperimento), si popola la struttura in prova e si esegue una sequenza di campionamenti delle operazioni `select(k)` e `rank(x)`. Prima di raccogliere i tempi utili viene effettuato un breve riscaldamento per ridurre l'effetto delle inizializzazioni (cache, allocator, ecc.).
Le singole misure sono raccolte ripetendo la stessa chiamata per ogni insieme di dati per ogni dimensione e registrando il tempo totale, da cui si ricava il tempo medio per dimensione; questi tempi medi vengono poi aggregati calcolando media e deviazione standard sull'insieme dei campioni e degli insiemi di dati per la stessa dimensione. Questo approccio riduce il rumore sperimentale e rende i risultati comparabili tra implementazioni.

3.7 StructureTester

Metodi Principali La classe `StructureTester` coordina l'intero esperimento. Progettata per essere parametrica e riutilizzabile, accetta in ingresso le dimensioni da testare, il numero di insiemi di dati per valore di n, il numero di ripetizioni da effettuare per ciascuna chiamata. La scelta di questi parametri influisce sul compromesso tra accuratezza statistica e tempo di esecuzione complessivo. La classe 'StructureTester' implementa una metodologia di test per ottenere misurazioni accurate e statisticamente significative. I metodi principali includono:

`plot_results(flag: Boolean, structure_names, random: Boolean )`
Genera grafici comparativi delle performance per le strutture specificate, distinguendo tra i tipi di insiemi di dati (casuali od ordinati) con il parametro `random`;
il parametro `flag` indica se si vuole visualizzare un grafico per ciascuna struttura dati singolarmente o se le strutture passate con `structure_names` devono essere confrontate in un unico grafico. flag = False genera un grafico per ogni struttura dati, flag = true genera un unico grafico di confronto tra le strutture dati passate in `structure_names`.

`run_tests(random: Boolean)` Esegue i test per tutte le strutture dati, raccogliendo i risultati in modo organizzato, il parametro `random` indica se utilizzare insiemi di dati casuali o ordinati.

Il processo è parametrizzato ed esegue:

- **Riscaldamento** (`warmup_calls`): Prima di ogni sessione di misurazione, vengono eseguite diverse chiamate a vuoto. Questo "riscaldamento" serve a stabilizzare l'ambiente di esecuzione, ad esempio popolando le cache della CPU ed evitando che l'overhead iniziale del sistema influenzi i risultati.
- **Iterazione su dimensioni diverse** (`sizes`): I test vengono eseguiti per un intervallo di dimensioni `n` (da 1 a 10.000), permettendo di osservare come le prestazioni scalano al crescere dei dati (questa scelta è stata fatta per non aumentare troppo il numero di insiemi di elementi per insiemi di dati e aumentando inutilmente il tempo di esecuzione dato che l'andamento risulta già chiaro).
- **Insiemi di dati multipli per dimensione** (`datasets_per_n`): Per ogni dimensione `n`, vengono generati più insiemi di dati indipendenti. I risultati vengono poi aggregati (calcolando media e deviazione standard). Questo riduce la probabilità che i risultati siano viziati da un singolo insieme di dati anomalo.

- **Campionamento delle operazioni** (`count`): Per ogni insieme di dati, le operazioni `select(k)` e `rank(x)` vengono testate con un numero elevato di valori in ingresso `k` e `x` scelti casualmente. Questo assicura che le prestazioni misurate siano rappresentative del comportamento medio dell'operazione sull'intera struttura.
- **Ripetizione delle chiamate** (`calls_per_test`): Ogni singola operazione (es. `select` con un `k` specifico) viene eseguita in un ciclo per un numero definito di volte. Il tempo totale viene poi diviso per il numero di chiamate, minimizzando il rumore e le fluttuazioni momentanee.

Alla fine del processo, 'StructureTester' aggrega tutti i campioni raccolti per ogni dimensione 'n' e calcola media e deviazione standard, che vengono poi utilizzate per generare i grafici di prestazione. I risultati finali sono conservati: per ogni dimensione si mantengono media e deviazione standard sia per `select` che per `rank`. StructureTester espone inoltre la possibilità di salvare i dati grezzi e di produrre grafici descrittivi con nomi di file costruiti automaticamente per facilitare l'analisi a posteriori.
StructureTester ha anche dei metodi che permettono di salvare i risultati usati per i grafici in tabelle latex.

Visualizzazione Per comodità d'uso sono incluse funzioni di plotting che costruiscono confronti tra le varie strutture e tra i tipi di dataset (random o degenerato). I grafici sono pensati per essere autoesplicativi: i nomi dei file incorporano le strutture protagoniste e il tipo di input, così che risultino immediatamente interpretabili.

4 Test effettuati

Sono stati effettuati due test principali per valutare le prestazioni delle tre strutture dati (`LinkedList`, `BSTree` senza `size`, `SBSTree` con `size`) nelle operazioni di `select` e `rank`. Di seguito sono descritti i test e i risultati ottenuti, con relativi andamenti sia singoli che comparativi.
Per i test i valori dei parametri della classe Tester sono stati scelti come segue:

- `warmup_calls`: 20
- `datasets_per_n`: 10
- `count`: 120
- `calls_per_test`: 30

4.1 Risultati dei test

Di seguito verranno mostrati i risultati per ogni test effettuato con le relative considerazioni.

4.1.1 Test 1: Dati Casuali

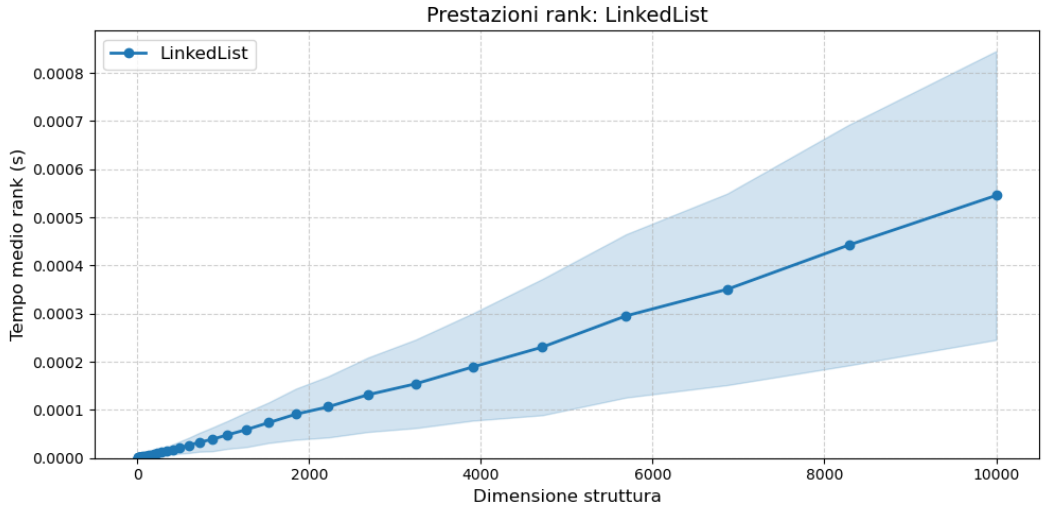


Figura 5: prestazioni Rank - LinkedList (random)

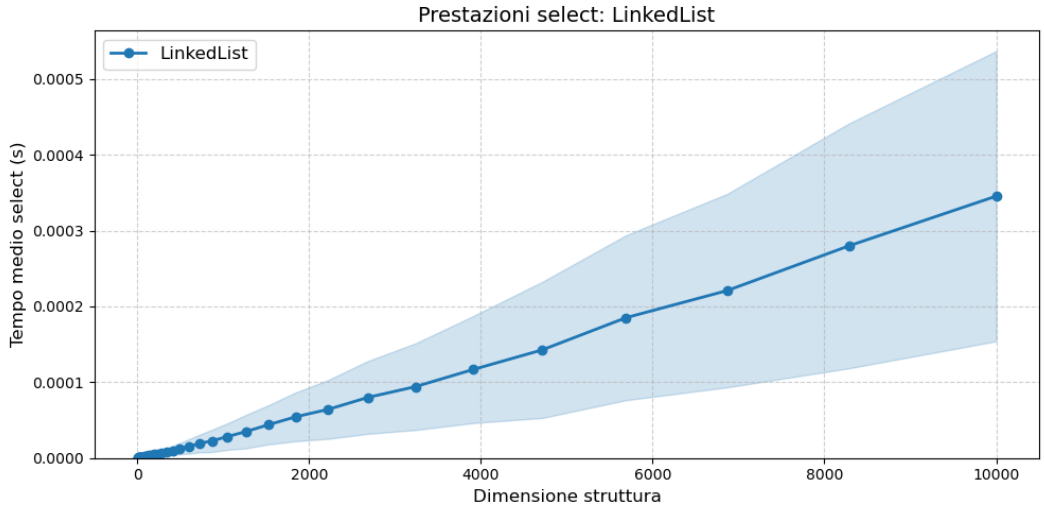


Figura 6: prestazioni Select - LinkedList (random)

`LinkedList`: O(n), lineare come previsto sia per select 5 che per rank 6.

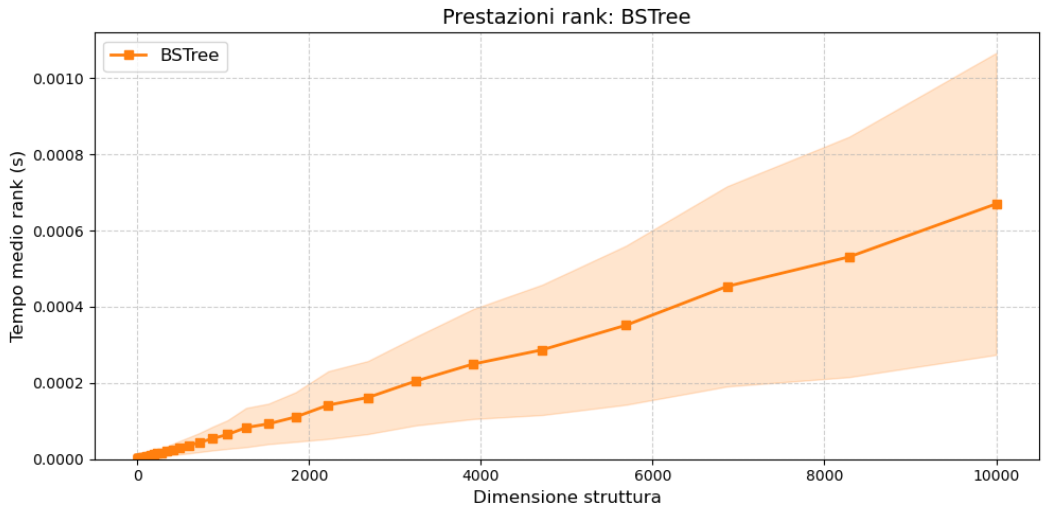


Figura 7: Rank - BSTree (random)

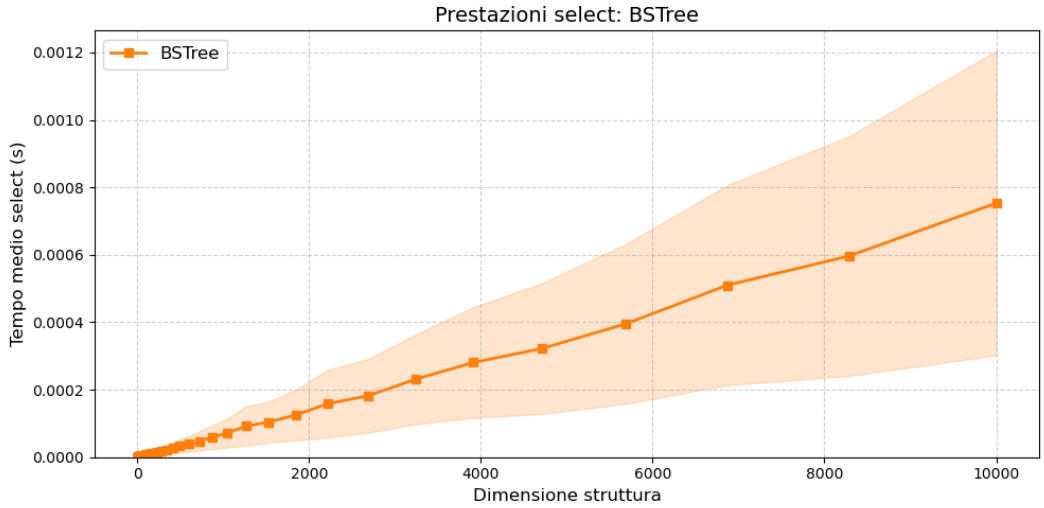


Figura 8: Select - BSTree (random)

Anche se i dati casuali producono mediamente un albero bilanciato con $h \approx \log n$, l'assenza di `size` costringe gli algoritmi select/rank a visitare/contare elementi. Risultato empirico atteso: le curve di tempo crescono linearmente con n 7,8, similmente a LinkedList (con costanti diverse).

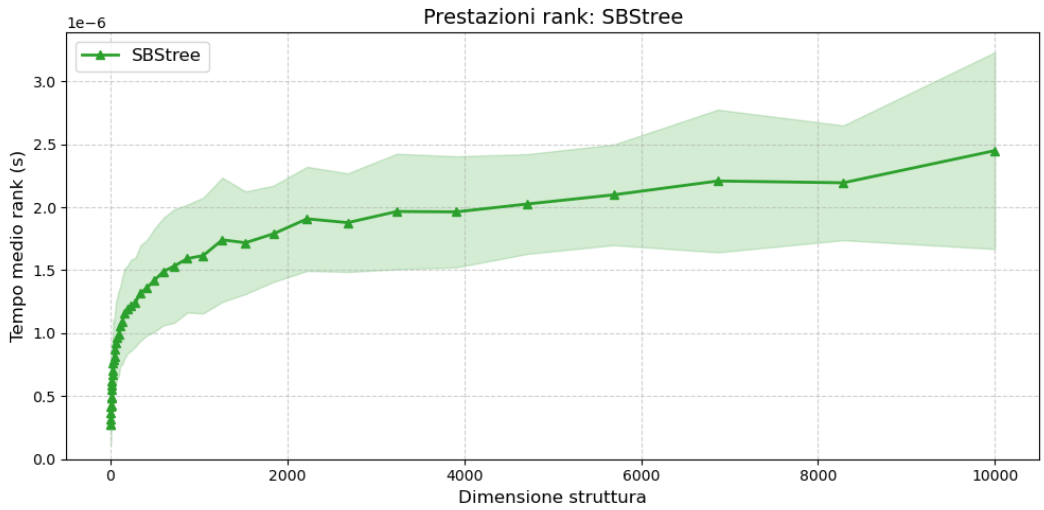


Figura 9: Rank - SBSTree (random)

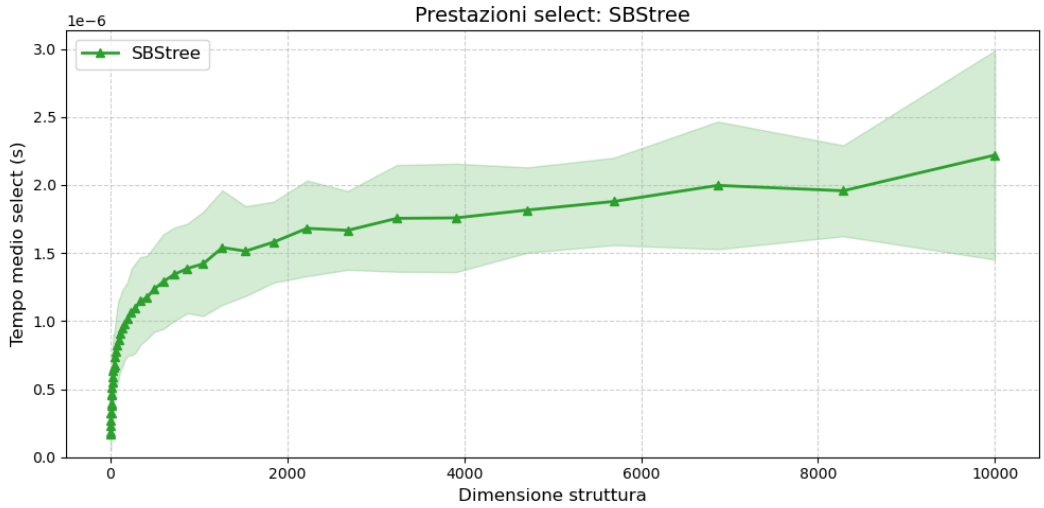


Figura 10: Select - SBSTree (random)

SBSTree: $O(\log n)$ come si vede da 9 e 10— grazie al bilanciamento AVL e all'attributo `size`.

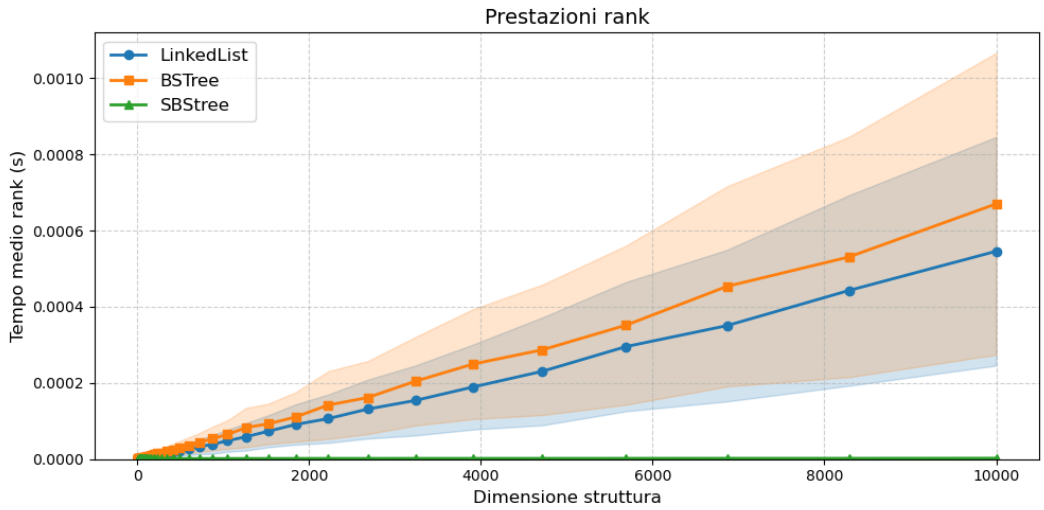


Figura 11: Confronto Rank (random)

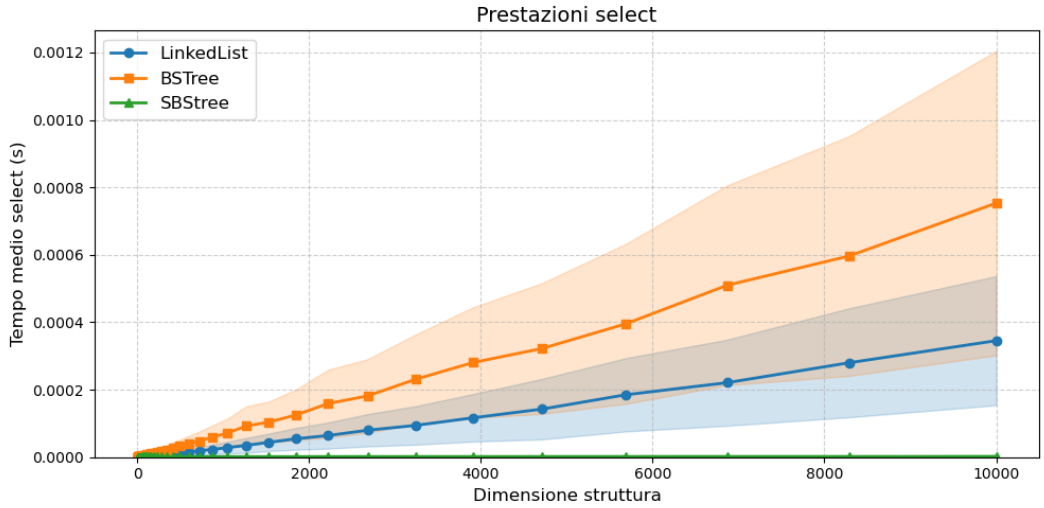


Figura 12: Confronto Select (random)

Conseguenza: il test dovrebbe mostrare che LinkedList e BSTree hanno prestazioni comparabili (entrambe $O(n)$), mentre SBSTree è nettamente migliore ($O(\log n)$).

Come si vede dalla tabella 2 e dai grafici comparativi 11, 12 i risultati confermano le aspettative teoriche:

LinkedList e BSTree: Come previsto dalla teoria, entrambe le strutture mostrano un tempo di esecuzione che cresce linearmente (O(n)) con la dimensione dell'input. Le loro curve di prestazioni sono quasi sovrapponibili, con il BSTree che mostra una costante leggermente più alta (pendenza maggiore) a causa dell'overhead della gestione della struttura ad albero rispetto alla semplice iterazione della lista.

SBSTree: Mostra prestazioni nettamente superiori. La sua curva è quasi piatta, confermando la complessità logaritmica (O(log n)). Il tempo di esecuzione cresce molto lentamente e non è quasi influenzato dall'aumento di n.

N	LinkedList		BSTree		SBStree	
	Select (s)	Rank (s)	Select (s)	Rank (s)	Select (s)	Rank (s)
1	8.291×10^{-8}	1.318×10^{-7}	2.674×10^{-7}	1.881×10^{-7}	1.656×10^{-7}	2.761×10^{-7}
4	1.378×10^{-7}	2.335×10^{-7}	3.997×10^{-7}	3.907×10^{-7}	2.714×10^{-7}	3.652×10^{-7}
7	2.088×10^{-7}	3.654×10^{-7}	5.942×10^{-7}	6.200×10^{-7}	3.763×10^{-7}	4.865×10^{-7}
13	3.576×10^{-7}	6.360×10^{-7}	9.846×10^{-7}	1.006×10^{-6}	4.655×10^{-7}	5.860×10^{-7}
24	6.366×10^{-7}	1.138×10^{-6}	1.792×10^{-6}	1.747×10^{-6}	5.903×10^{-7}	7.038×10^{-7}
42	1.053×10^{-6}	1.896×10^{-6}	2.966×10^{-6}	2.897×10^{-6}	6.781×10^{-7}	8.131×10^{-7}
75	1.826×10^{-6}	3.283×10^{-6}	5.091×10^{-6}	4.920×10^{-6}	8.201×10^{-7}	9.638×10^{-7}
132	3.197×10^{-6}	5.742×10^{-6}	8.839×10^{-6}	8.448×10^{-6}	9.454×10^{-7}	1.095×10^{-6}
232	5.682×10^{-6}	1.023×10^{-5}	1.549×10^{-5}	1.471×10^{-5}	1.063×10^{-6}	1.222×10^{-6}
409	1.006×10^{-5}	1.786×10^{-5}	2.743×10^{-5}	2.564×10^{-5}	1.173×10^{-6}	1.359×10^{-6}
719	1.914×10^{-5}	3.284×10^{-5}	4.763×10^{-5}	4.359×10^{-5}	1.343×10^{-6}	1.533×10^{-6}
1264	3.495×10^{-5}	5.884×10^{-5}	9.227×10^{-5}	8.300×10^{-5}	1.541×10^{-6}	1.741×10^{-6}
2222	6.424×10^{-5}	1.064×10^{-4}	1.591×10^{-4}	1.420×10^{-4}	1.682×10^{-6}	1.908×10^{-6}
3906	1.167×10^{-4}	1.893×10^{-4}	2.805×10^{-4}	2.495×10^{-4}	1.759×10^{-6}	1.963×10^{-6}
6866	2.208×10^{-4}	3.505×10^{-4}	5.097×10^{-4}	4.534×10^{-4}	1.997×10^{-6}	2.208×10^{-6}
10000	3.454×10^{-4}	5.456×10^{-4}	7.533×10^{-4}	6.699×10^{-4}	2.220×10^{-6}	2.449×10^{-6}

Tabella 2: Tempi medi per dataset random (Cifre sign.: 3).

4.1.2 Test 2: Dati Ordinati (Albero BSTree Degenere)

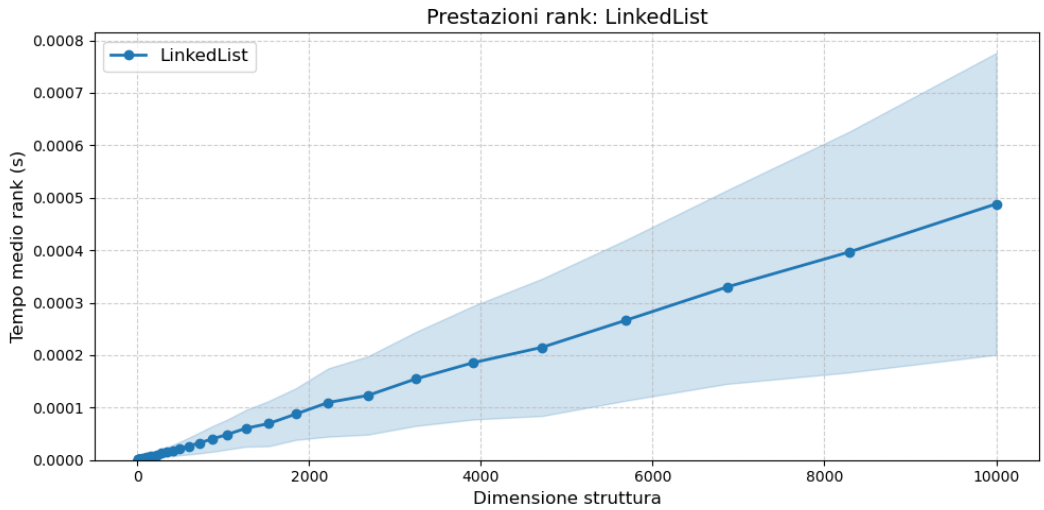


Figura 13: Rank - LinkedList (ordinato)

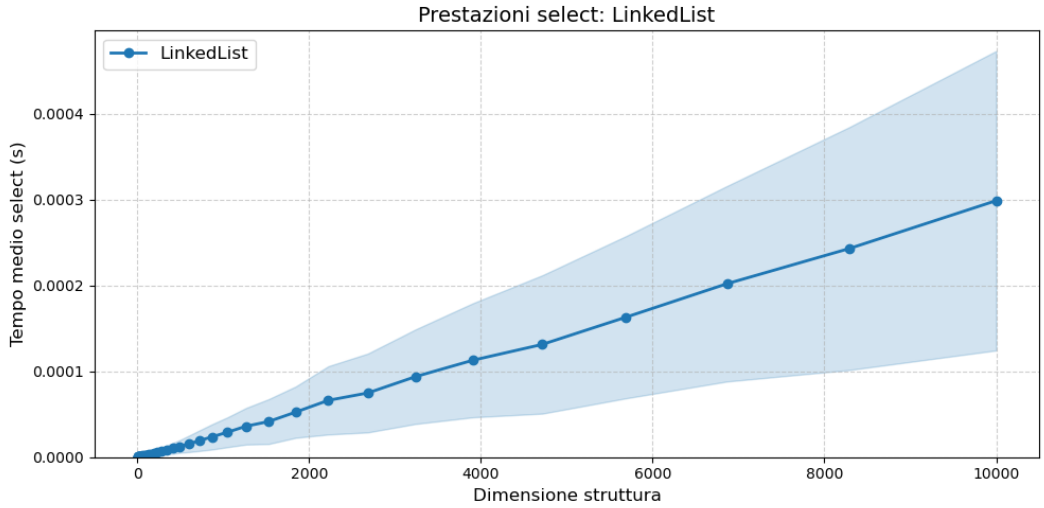


Figura 14: Select - LinkedList (ordinato)

LinkedList: O(n) (lineare) simile al test precedente, dato che la lista mantiene la stessa struttura indipendentemente dall'ordine di inserimento 13, 14.

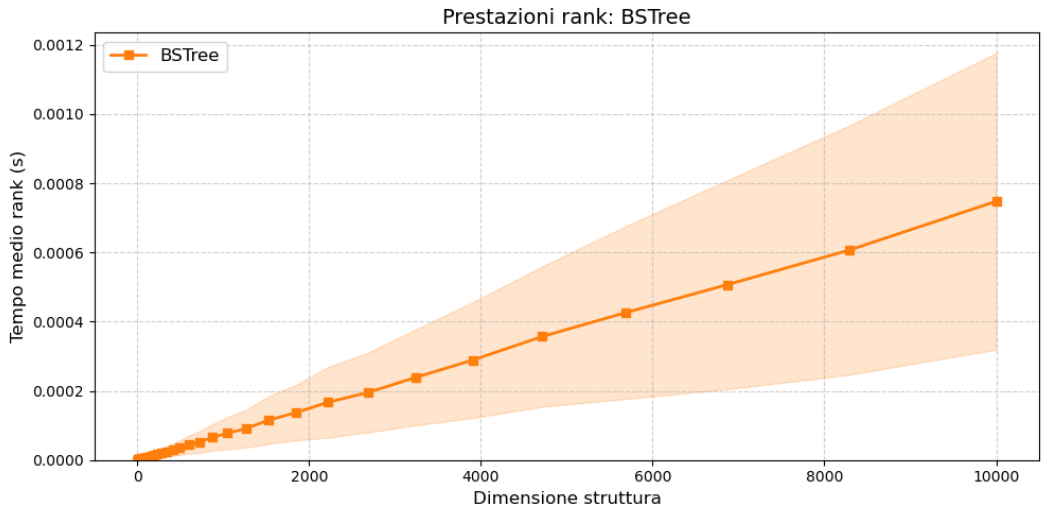


Figura 15: Rank - BSTree (ordinato)

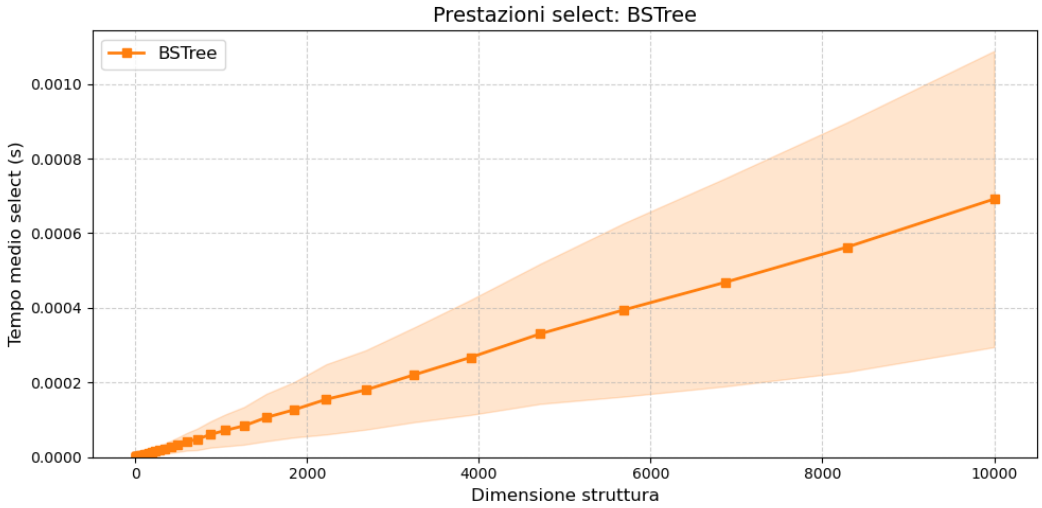


Figura 16: Select - BSTree (ordinato)

BSTree (degenere): O(n) simile al test precedente 15, 16.

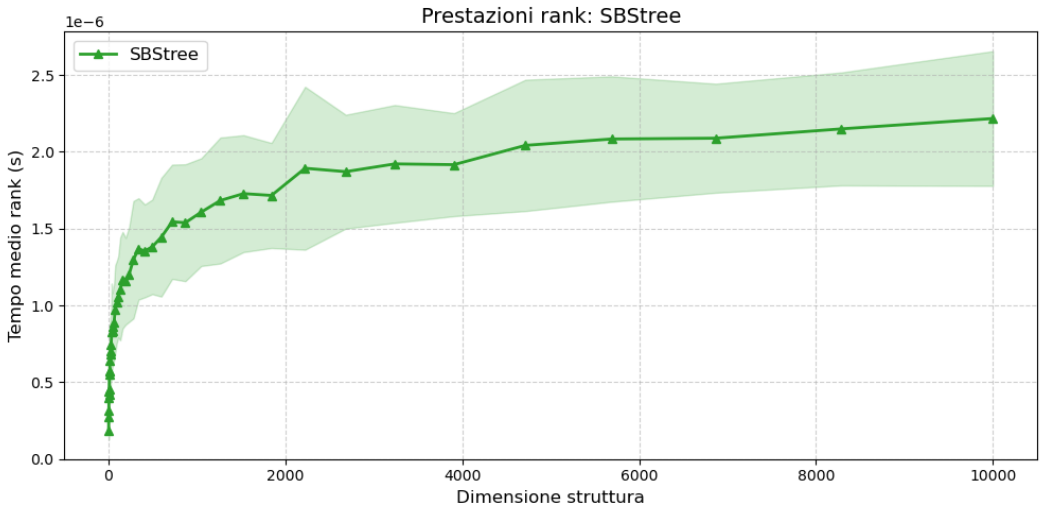


Figura 17: Rank - SBSTree (ordinato)

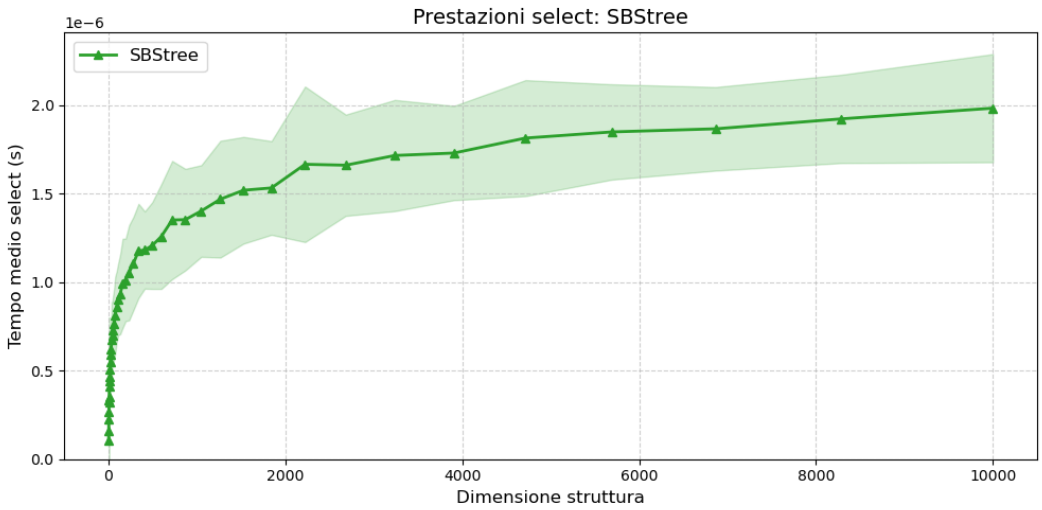


Figura 18: Select - SBSTree (ordinato)

SBSTree: $O(\log n)$ — bilanciamento AVL mantiene $h = O(\log n)$, 17, 18.

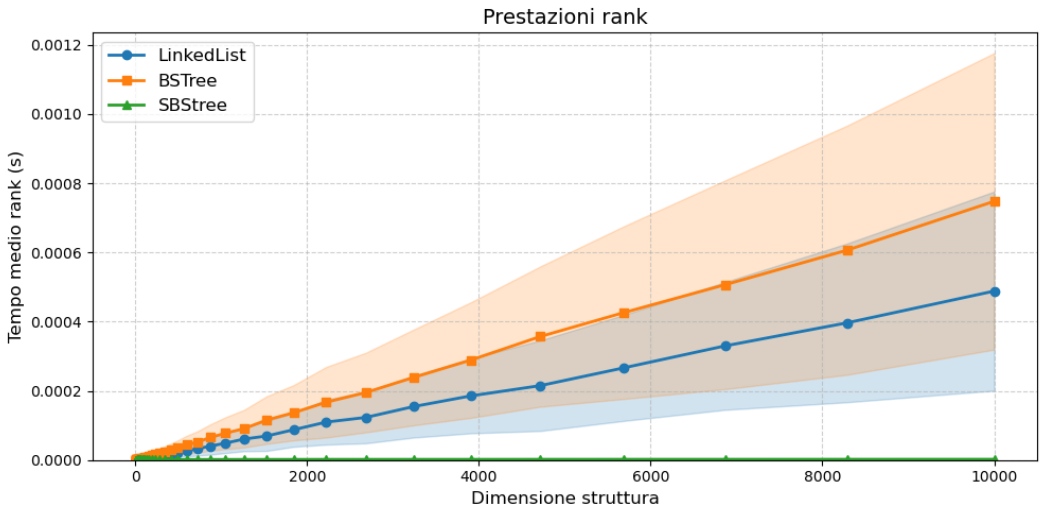


Figura 19: Confronto Rank (ordinato)

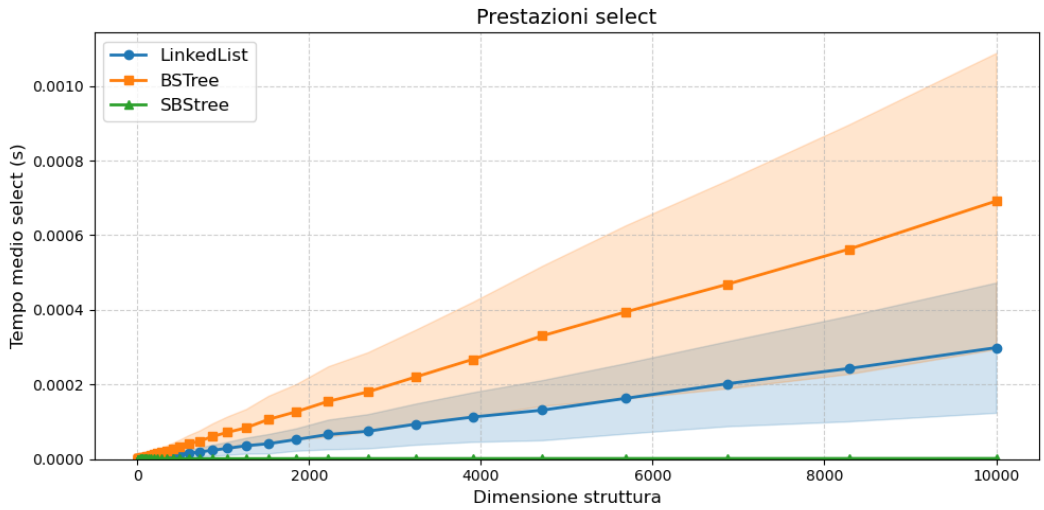


Figura 20: Confronto Select (ordinato)

N	LinkedList		BSTree		SBStree	
	Select (s)	Rank (s)	Select (s)	Rank (s)	Select (s)	Rank (s)
1	6.185×10^{-8}	9.787×10^{-8}	1.823×10^{-7}	1.293×10^{-7}	1.081×10^{-7}	1.864×10^{-7}
4	1.345×10^{-7}	2.373×10^{-7}	3.928×10^{-7}	3.503×10^{-7}	2.651×10^{-7}	3.969×10^{-7}
7	2.178×10^{-7}	3.817×10^{-7}	6.095×10^{-7}	5.826×10^{-7}	3.523×10^{-7}	4.523×10^{-7}
13	3.554×10^{-7}	6.294×10^{-7}	1.015×10^{-6}	1.017×10^{-6}	4.651×10^{-7}	5.711×10^{-7}
24	6.327×10^{-7}	1.132×10^{-6}	1.766×10^{-6}	1.840×10^{-6}	5.908×10^{-7}	7.018×10^{-7}
42	1.080×10^{-6}	1.927×10^{-6}	2.906×10^{-6}	3.086×10^{-6}	7.010×10^{-7}	8.359×10^{-7}
75	1.909×10^{-6}	3.424×10^{-6}	5.253×10^{-6}	5.682×10^{-6}	8.130×10^{-7}	9.736×10^{-7}
132	3.283×10^{-6}	5.912×10^{-6}	9.008×10^{-6}	9.764×10^{-6}	9.321×10^{-7}	1.107×10^{-6}
232	5.598×10^{-6}	1.007×10^{-5}	1.543×10^{-5}	1.677×10^{-5}	1.053×10^{-6}	1.201×10^{-6}
409	1.019×10^{-5}	1.800×10^{-5}	2.751×10^{-5}	2.985×10^{-5}	1.181×10^{-6}	1.355×10^{-6}
719	1.918×10^{-5}	3.255×10^{-5}	4.789×10^{-5}	5.173×10^{-5}	1.351×10^{-6}	1.544×10^{-6}
1264	3.606×10^{-5}	6.063×10^{-5}	8.407×10^{-5}	9.079×10^{-5}	1.468×10^{-6}	1.683×10^{-6}
2222	6.620×10^{-5}	1.098×10^{-4}	1.550×10^{-4}	1.672×10^{-4}	1.666×10^{-6}	1.893×10^{-6}
3906	1.129×10^{-4}	1.854×10^{-4}	2.676×10^{-4}	2.891×10^{-4}	1.729×10^{-6}	1.916×10^{-6}
6866	2.021×10^{-4}	3.299×10^{-4}	4.684×10^{-4}	5.066×10^{-4}	1.866×10^{-6}	2.088×10^{-6}
10000	2.988×10^{-4}	4.882×10^{-4}	6.916×10^{-4}	7.476×10^{-4}	1.983×10^{-6}	2.216×10^{-6}

Tabella 3: Tempi medi per dataset sorted (Cifre sign.: 3).

Come si vede dalla tabella 3 e dai grafici comparativi 19, 20 i risultati confermano le aspettative teoriche:
LinkedList e BSTree: in questo scenario, il **BSTree** degenera in una lista.
SBSTree: grazie al meccanismo di autobilanciamento AVL, la struttura non degenera, mantiene un'altezza logaritmica e le sue prestazioni per select e rank rimangono **O(log n)**, come evidenziato dalla curva piatta nei grafici.

5 Conclusione

In questa relazione sono state confrontate le prestazioni pratiche di tre implementazioni per le statistiche d'ordine dinamiche: `LinkedList`, `BSTree` (albero binario non bilanciato senza campo `size`) e `SBStree` (albero AVL che mantiene il campo `size`).

Riepilogo dei risultati:

- **SBStree** fornisce le migliori prestazioni per entrambe le operazioni `select` e `rank` nelle misure presentate, con crescita logaritmica apparente e varianza contenuta.
- **BSTree** (non bilanciato e senza campo `size`) mostra ampia variabilità specialmente per `rank`; i tempi possono variare molto tra istanze diverse, non dipendono dall' altezza.

- **LinkedList** ha comportamento prevedibile (linearità per select/rank) ma non è competitiva su grandi dimensioni rispetto alla struttura con `size`.

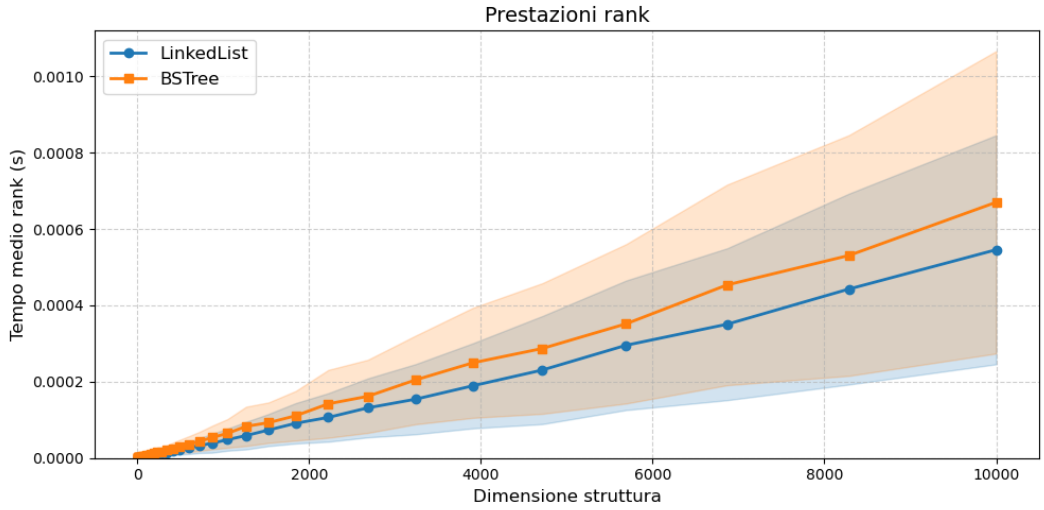
5.1 Conclusione test 1

Conclusione Test 1:viene rispettata l'attesa teorica come evidenziato da 2 Anche in uno scenario medio con dati casuali, l'assenza dell'attributo `size` rende il `BSTree` inefficiente quanto una `LinkedList`. L'`SBSTree` è di gran lunga la soluzione più performante.

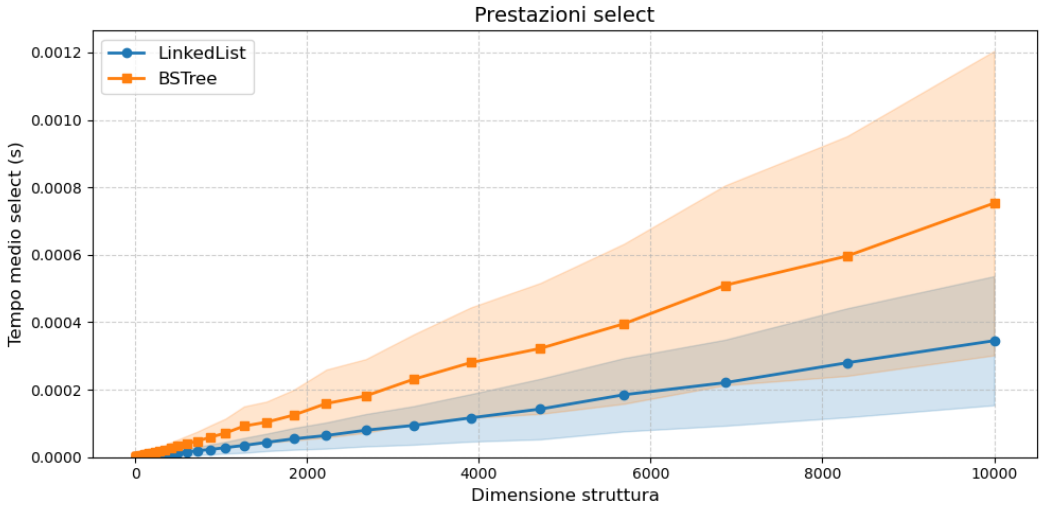
5.2 Conclusione 2

Conclusione Test 2: Questo test evidenzia la non importanza del bilanciamento se manca l'attributo size, in quanto i risultati sono simili ai precedenti; anche se l'albero fosse perfettamente bilanciato ($h = O(\log n)$), l'assenza di informazioni pre-calcolate sulla dimensione dei sottoalberi costringe l'algoritmo a visitare/contare elementi, rendendo il costo $O(n)$ indipendentemente dal bilanciamento mentre, se vi fossero informazioni pre calcolate, il costo dipenderebbe anche per **BSTree** dall' altezza diventando soggetto all' ordine dei dati, ma non essendo autobilanciato risulterebbe meno stabile di **SBSTree**.

Analisi delle Pendenze Analisi delle Pendenze (O(n)): LinkedList vs. BSTree



figureRank performance (overview)



figureRank performance (overview)

- **Per select(k):**
  - Il grafico mostra che `BSTree` ha una pendenza significativamente maggiore (più ripida) rispetto a `LinkedList`; entrambe hanno complessità media $O(n)$ ma con costanti diverse.
  - `LinkedList` esegue una semplice scansione sequenziale (`current = current.next`), operazione molto efficiente a basso livello.
  - `BSTree` richiede una visita in-order (iterativa o ricorsiva) che implica la gestione di uno stack e l'attraversamento dei puntatori `left`/`right`, comportando un maggiore overhead per nodo visitato.
- **Per rank(x):**
  - I grafici mostrano pendenze molto simili per le due strutture.
  - `LinkedList` esegue una scansione lineare contando i nodi (costo $O(n)$).
  - `BSTree` esegue discesa e invoca funzioni di conteggio sui sottoalberi (ad es. `_subtree_size`); nonostante le chiamate ricorsive, il numero totale di nodi visitati risulta "comparabile" a quello della scansione lineare.

In sintesi, i test confermano che per `select` la scansione della lista ha costanti molto favorevoli rispetto alla visita in-order dell'albero, mentre per `rank` i costi totali risultano comparabili tra le due strutture.

5.3 Ambiente di Sviluppo

I test sono stati eseguiti in un ambiente di sviluppo controllato per garantire la riproducibilità e l'affidabilità dei risultati. Di seguito sono riportati i dettagli dell'ambiente hardware e software utilizzato:

- **Hardware:**
  - Processore: Intel Core i7-8565U × 8
  - RAM: 8GB DDR4
  - Sistema Operativo: Ubuntu 24.04.3 LTS con Kernel Linux 6.8.0-85-generic

per l'ambiente si è usato conda e come ide pycharm, per l'UML si è usato starUML, per la relazione si è usata un' estensione di pycharm per latex.

6 Bibliografia

Riferimenti bibliografici

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, Livio Colussi, *Introduzione agli algoritmi e strutture dati*, Collana di istruzione scientifica. Serie di informatica, McGrawHill Companies, 2010.

oqfno