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 **1**

2 Descrizione del problema e implementazione **1**
2.1 Statistiche d'Ordine Dinamico 1
2.2 Complessità Teorica 2
2.2.1 Lista Concatenata Ordinata 2
2.2.2 Albero Binario di Ricerca Standard 2
2.2.3 Albero Binario di Ricerca con Attributo Dimensione 2

3 Documentazione **2**
3.1 Struttura del Progetto 2
3.2 LinkedList 2
3.3 BSTree 3
3.4 SBSTree 4
3.5 Framework di Test 4

1 Introduzione

Questa relazione analizza le prestazioni di diverse strutture dati nell'implementazione delle statistiche d'ordine dinamico, in particolare le operazioni di `select(k)` e `rank(x)`. Queste operazioni rappresentano problemi fondamentali nell'ambito delle strutture dati dinamiche, con applicazioni pratiche in database, sistemi di elaborazione delle query e algoritmi di ordinamento.

Lo scopo di questa analisi è confrontare tre implementazioni differenti:

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

Verranno presentati i risultati sperimentali che evidenziano come le diverse implementazioni si comportano al variare delle dimensioni del dataset, con particolare attenzione a efficienza temporale e variabilità delle prestazioni.

2 Descrizione del problema e implementazione

2.1 Statistiche d'Ordine Dinamico

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.

2.2 Complessità Teorica

2.2.1 Lista Concatenata Ordinata

In una lista concatenata ordinata:

- `select(k)` richiede O(k) passaggi, poiché è necessario attraversare la lista dall'inizio fino al k-esimo elemento.
- `rank(x)` richiede O(n) passaggi nel caso peggiore, dove n è la dimensione della lista, poiché potrebbe essere necessario scorrere l'intera lista.

2.2.2 Albero Binario di Ricerca Standard

In un albero binario di ricerca (BST) non bilanciato:

- `select(k)` richiede O(n) nel caso peggiore, poiché l'implementazione tradizionale richiede una traversata dell'albero che può coinvolgere potenzialmente tutti i nodi.
- `rank(x)` richiede O(n) nel caso peggiore, per le stesse ragioni.

Tuttavia, in un albero bilanciato, entrambe le operazioni potrebbero raggiungere O(log n), ma questo non è garantito nell'implementazione standard senza meccanismi di bilanciamento.

2.2.3 Albero Binario di Ricerca con Attributo Dimensione

Un BST con campo `size` che mantiene la dimensione del sottoalbero di ogni nodo:

- `select(k)` richiede O(h) operazioni, dove h è l'altezza dell'albero. Nei casi peggiori di alberi sbilanciati, h può essere O(n), ma in alberi ragionevolmente bilanciati h si avvicina a O(log n).
- `rank(x)` richiede anche O(h), con le stesse considerazioni.

Questa implementazione offre significativi miglioramenti teorici rispetto alle alternative, specialmente quando l'albero non degenera in una lista.

3 Documentazione

3.1 Struttura del Progetto

Il progetto è organizzato nelle seguenti componenti principali:

- `src/DataStracture/LinkedList/`: Implementazione della lista concatenata ordinata
- `src/DataStracture/BSTree/`: Implementazione dell'albero binario di ricerca standard
- `src/DataStracture/SBSTree/`: Implementazione dell'albero binario di ricerca con attributo dimensione
- `src/DataGenerator.py`: Generatore di dati casuali per i test
- `src/Tester.py`: Framework per i test di prestazioni
- `src/main.py`: Script principale per l'esecuzione dei test

3.2 LinkedList

La classe `LinkedList` implementa una lista concatenata ordinata. Gli elementi sono mantenuti in ordine crescente durante l'inserimento.

```
def select(self, k) -> Node or None:
    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) -> int or None:
    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.

```
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, operazione potenzialmente costosa quando invocata frequentemente.

3.4 SBSTree

La classe `SBSTree` implementa un albero binario di ricerca dove ogni nodo mantiene la dimensione del suo sottoalbero.

```
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 sottoalberi, consentendo operazioni più efficienti.

3.5 Framework di Test

Il framework di test è stato progettato per valutare rigorosamente le prestazioni delle diverse implementazioni:

- **Generazione dati**: Ogni test genera dataset di dimensione crescente
- **Warm-up**: Prima delle misurazioni vengono effettuate chiamate di warm-up per stabilizzare la cache e ridurre l'overhead iniziale
- **Campionamento**: Per ogni dataset vengono estratti casualmente indici k da testare
- **Ripetizioni**: Ogni operazione viene ripetuta più volte per ridurre il rumore di misurazione
- **Statistiche**: I risultati sono aggregati calcolando media e deviazione standard dei tempi

4 test effettuati

4.1 Metodologia

I test sono stati eseguiti con i seguenti parametri:

- Dimensione delle strutture dati: da 100 a 10000 elementi
- Dataset generati per ogni dimensione: 5
- Campioni per dataset: 50
- Ripetizioni per misurazione: 300
- Chiamate di warm-up: 30
- Seme casuale fisso: 42

4.2 Analisi dell'operazione rank()

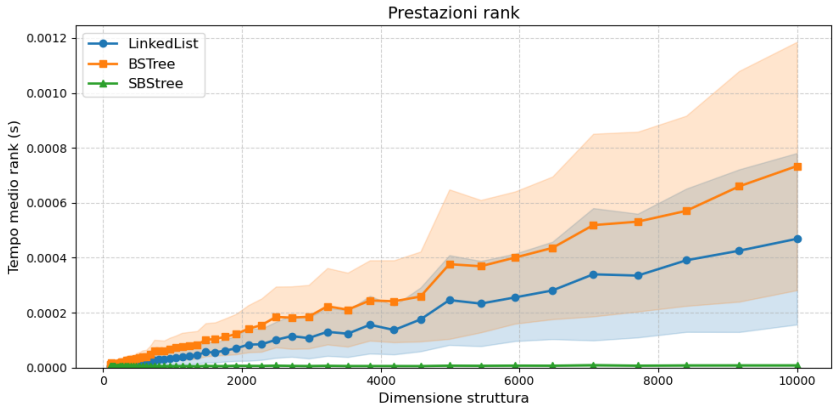

Figura 1: Rank performance

Figura 2: Confronto delle prestazioni di rank() tra le strutture

Osservando il grafico di Figura **??**, si evidenziano i seguenti comportamenti:

- **SBSTree** (linea verde) mostra prestazioni nettamente superiori, con tempi quasi costanti (nell'ordine di 10^{-5} secondi) al crescere della dimensione, confermando il comportamento teorico logaritmico.
- **LinkedList** (linea blu) mostra una crescita lineare con la dimensione della struttura, ma con una pendenza moderata e una variabilità relativamente bassa, indicata dalla banda di deviazione standard stretta.
- **BSTree** (linea arancione) mostra le prestazioni peggiori con tempi di esecuzione più elevati e, soprattutto, una variabilità molto ampia (evidenziata dalla banda di deviazione standard larga). Questo è dovuto all'implementazione che richiede il calcolo ricorsivo della dimensione dei sottoalberi ad ogni chiamata, senza memorizzazione.

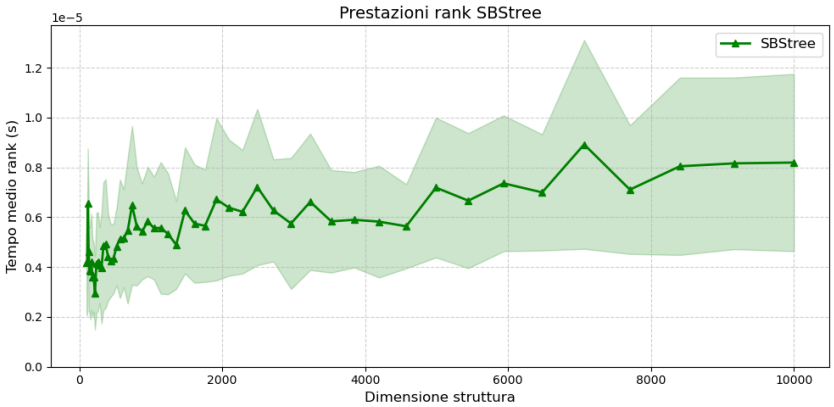

Figura 3:

Figura 4: Dettaglio delle prestazioni di rank() per SBSTree

Il grafico in Figura **??** evidenzia in dettaglio le prestazioni di `SBSTree`:

- Il tempo di esecuzione rimane estremamente basso (nell'ordine di 10^{-5} secondi).
- Si nota una leggera crescita logaritmica al crescere della dimensione, coerente con la teoria.
- La variabilità (banda verde) è significativa in proporzione al valore medio, indicando sensibilità alla forma specifica dell'albero e alla posizione dell'elemento cercato.
- Il grafico mostra un plateau dopo la fase iniziale, suggerendo che gli effetti di cache e altri overhead di sistema diventano predominanti rispetto al costo algoritmico dell'operazione.

4.3 Analisi dell'operazione select()

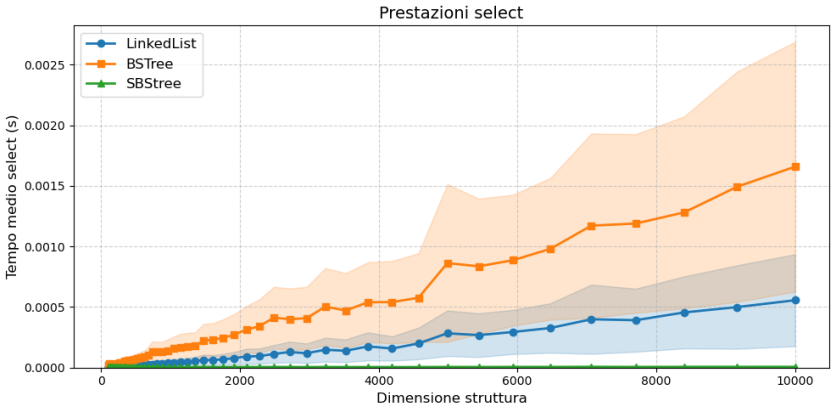

Figura 5: Confronto delle prestazioni di select() tra le strutture

Analizzando il grafico in Figura **??**:

- **SBSTree** (linea verde) conferma la sua superiorità anche per l'operazione `select()`, con tempi di esecuzione prossimi allo zero anche per grandi dimensioni.
- **LinkedList** (linea blu) mostra una crescita lineare, ma con tempi mediamente inferiori rispetto al `BSTree`. Questo può sembrare controintuitivo, ma è spiegabile con l'overhead più basso delle operazioni di scorrimento lineare in memoria rispetto alla ricorsione necessaria per l'albero senza attributo dimensione.
- **BSTree** (linea arancione) mostra nuovamente le prestazioni peggiori con tempi elevati e alta variabilità. L'implementazione attraverso visita in-order fino al k-esimo elemento risulta inefficiente, specialmente per valori di k elevati.

5 Discussione

5.1 Prestazioni Comparative

I risultati sperimentali confermano le previsioni teoriche:

- **SBSTree** è nettamente la struttura più efficiente per entrambe le operazioni, con prestazioni logaritmiche e bassa variabilità.
- **LinkedList** offre prestazioni lineari ma prevedibili. Per piccole dimensioni, la sua semplicità può renderla competitiva rispetto al `BSTree` standard.
- **BSTree** senza attributo dimensione mostra le prestazioni peggiori, sia in termini di tempi medi che di variabilità. La necessità di ricalcolare ricorsivamente le dimensioni dei sottoalberi ad ogni chiamata di `rank()` e la visita completa fino al k-esimo elemento per `select()` lo rendono inefficiente.

5.2 Variabilità e Stabilità

Un aspetto importante evidenziato dai risultati è la variabilità delle prestazioni:

- Il `BSTree` standard mostra la più alta variabilità, con un'ampia banda di deviazione standard. Questo è attribuibile alla forma potenzialmente sbilanciata dell'albero e alla dipendenza del tempo di esecuzione dalla posizione specifica dell'elemento cercato.
- La `LinkedList` mostra una variabilità moderata, principalmente influenzata dalla posizione dell'elemento cercato.
- Il `SBSTree`, pur avendo la variabilità più bassa in termini assoluti, mostra una certa variabilità relativa al suo valore medio molto basso. Questo è dovuto principalmente all'influenza della cache e di altri fattori di sistema su operazioni così veloci.

5.3 Effetto della Dimensione del Dataset

L'analisi dei grafici evidenzia chiaramente come la dimensione del dataset influenzi le prestazioni:

- La `LinkedList` e il `BSTree` mostrano una crescita significativa dei tempi al crescere della dimensione, confermando il loro comportamento O(n).
- Il `SBSTree` mantiene tempi quasi costanti anche per grandi dimensioni, dimostrando il suo comportamento logaritmico O(log n).
- Per dimensioni molto piccole (¡ 1000 elementi), le differenze tra le strutture sono meno pronunciate, suggerendo che per piccoli dataset la scelta della struttura potrebbe essere meno critica.

5.4 Implicazioni Pratiche

I risultati hanno importanti implicazioni pratiche:

- Per applicazioni che richiedono frequenti operazioni di `select()` e `rank()` su dataset dinamici, l'utilizzo di una struttura con attributo dimensione come `SBSTree` è fortemente consigliato.
- Il costo aggiuntivo di memoria per memorizzare la dimensione in ogni nodo è ampiamente compensato dal guadagno in prestazioni.
- Per dataset piccoli o applicazioni dove la semplicità è prioritaria, una `LinkedList` ordinata può essere una soluzione ragionevole.
- L'implementazione standard di `BSTree` senza attributo dimensione non è consigliabile per applicazioni che richiedono frequenti operazioni di statistiche d'ordine.

6 Conclusioni

Questa analisi ha confrontato tre implementazioni delle statistiche d'ordine dinamico, evidenziando come l'aggiunta di informazioni supplementari (l'attributo dimensione nei nodi dell'albero) possa migliorare drasticamente le prestazioni.

Il `SBSTree` si è dimostrato nettamente superiore sia per l'operazione `select(k)` che per `rank(x)`, con tempi di esecuzione logaritmici e bassa variabilità anche su dataset di grandi dimensioni.

La `LinkedList`, nonostante la sua semplicità e il comportamento lineare, ha mostrato prestazioni sorprendentemente buone, superando il `BSTree` standard in molti casi grazie alla sua implementazione diretta e all'assenza di overhead ricorsivi.

Il `BSTree` standard ha mostrato le prestazioni peggiori, evidenziando come un'implementazione naive delle statistiche d'ordine su alberi binari di ricerca sia inefficiente quando non si memorizzano informazioni aggiuntive sulla struttura.

Per applicazioni che richiedono frequenti operazioni di statistiche d'ordine dinamico, l'investimento in strutture dati più sofisticate come il `SBSTree` risulta ampiamente giustificato dai significativi guadagni in termini di prestazioni.