UNIVERSITÀ
DEGLI STUDI
FIRENZE

UNIVERSITÀ DEGLI STUDI DI FIRENZE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Laboratorio di Algoritmi e Strutture Dati

Autore:
Jacopo Mechi

N° Matricola:
7047035

Corso principale:
Algoritmi e Strutture Dati

Docente corso:
Simone Marinai

Indice

Elenco delle figure

1 Introduzione generale

1.1 Progetto asseganto

Chiavi duplicate in alberi binari di ricerca

1.2 Breve descrizione dello svolgimento del esercizio

Suddivideremo la descrizione del esercizio in 4 parti fondamentali:

- **Spiegazione teorica del problema** : è dove si descrive il problema che andremo ad affrontare in modo teorico partendo dagli assunti del libro di Algoritmi e Strutture Dati.
- **Documentazione del codice** : in questa parte spieghiamo come il codice del esercizio viene implementato
- **Descrizione degli esperimenti condotti** : partendo dal codice ed effettuando varie misurazioni cerchiamo di verificare le ipotesi teoriche
- **Analisi dei risultati sperimentali** : dopo aver svolto gli esperimenti riflettiamo risultati ed esponiamo una tesi

1.3 Specifiche della piattaforma di test

- **CPU** : AMD Ryzen 7 5800X3D 3.400GHz 8 core 16 thread
- **RAM** : Crucial Ballistix 16GB DDR4 3600MHz
- **SSD** : Samsung 850 EVO 250 GB SATA III

Il linguaggio di programmazione utilizzato sarà Python, la piattaforma in cui il codice è stato scritto è il text editor **NVIM v0.9.0** e ”girato” sulla shell **zsh 5.9**. La stesura di questo testo è avvenuta con le stesse modalità.

Parte I
Chiavi duplicate in alberi binari di ricerca

Esercizio

- Vogliamo confrontare vari modi per gestire chiavi duplicate in ABR:
  - implementazione ”normale” (senza accorgimenti particolari)
  - utilizzando un flag booleano
  - mantenendo una lista di nodi con chiavi uguali
- Per fare questo dovremo:
  - Scrivere i programmi Python (no notebook) che:
    - implementano 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 del problema

2.1 Introduzione

In questa sezione vedremo una rapida infarinatura teorica sugli alberi binari di ricerca e sulle operazioni di inserimento e di ricerca. Per i nostri esperimenti utilizzeremo queste due operazioni poiché il codice del inserimento lo avremo necessariamente implementato per popolare gli alberi, mentre utilizzeremo l'operazione di ricerca poiché alla base di ogni operazione eseguibile su un ABR c'è un cammino semplice e quindi fondamentalmente una ricerca (compreso l'inserimento) e quindi i tempi di esecuzione avranno lo stesso andamento per qualsiasi operazione.

2.2 Aspetti fondamentali

Un albero binario di ricerca (esempio in figura 1) è una tipologia particolare di albero binario con le seguenti caratteristiche:

1. Il sottoalbero sinistro di un nodo x contiene soltanto i nodi con chiavi minori della chiave del nodo x.
2. Il sottoalbero destro di un nodo x contiene soltanto i nodi con chiavi maggiori della chiave del nodo x.
3. Il sottoalbero destro e il sottoalbero sinistro devono essere entrambi due ABR.

Descriviamo ora le caratteristiche delle due operazioni che prenderemo in considerazione:

- Per l'inserimento, iniziando dalla radice dell'albero, si sceglie ricorsivamente su quale ramo spostarsi basandoci sul confronto tra la chiave della foglia in cui siamo e il valore che vogliamo inserire. Arrivati in fondo all'albero creeremo un nuovo nodo, se destro o sinistro rispetto all'ultima foglia dipende dall'implementazione. Vediamo ora la differenza tra le varie implementazioni che andremo ad analizzare, tenendo a mente che esse si differenziano solo per come gestiscono le chiavi duplicate mentre hanno lo stesso comportamento negli altri casi (inserimento destro se la chiave del nuovo nodo é maggiore e sinistro se la chiave é minore).
  - Classica: la chiave duplicata viene sempre inserita a destra
  - Flag booleana: ogni nodo contiene una flag booleana che cambia il suo valore ogni volta che viene visitata da una chiave duplicata. Sinistra se la flag è True e destra se False

  - Lista concatenta di nodi con chiavi uguali: ogni nodo viene trasformato in una lista concatenata con inserimento in testa per salvare le chiavi duplicate

- Nell' operazione di ricerca per ogni foglia, prima confrontiamo il valore che stiamo cercando con quello della suddetta foglia. Se il valore è uguale allora ritorneremo un boolean di conferma. Se raggiunta l'ultima foglia dell'albero non trovassimo il valore allora ritorneremo un boolean di fallimento. Altrimenti se non ci troviamo in nessuna delle due situazioni appena descritte scendiamo l'albero con lo stesso metodo dell'inserimento. Ovviamente per fare la ricerca partiremo dalla radice dell'albero.

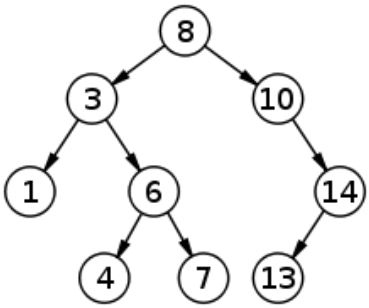

Figura 1: Albero binario di ricerca

2.3 Assunti ed ipotesi

In un ABR le operazioni di base richiedono un tempo proporzionale all'altezza dell'albero. L'altezza attesa di un ABR costruito in modo casuale è $O(h = lg(n))$ quindi le operazioni elementari svolte su questo tipo di albero richiedono in media $\Theta(h)$. Nel caso peggiore, il caso in cui l'albero sia completamente sbilanciato da un lato, dando cosi origine ad una lista, l'altezza è $\Theta(n)$ e quindi ci aspettiamo che le operazioni elementari richiedano $\Theta(n)$ per essere svolte. Per vedere la complessità degli algoritmi più importanti di ABR basati sul caso peggiore e sul caso medio si richiama alla figura 2 facendo particolare attenzione ai metodi in rosso che sono quelli su cui andremo a svolgere gli esperimenti. Poiché andremo a sperimentare su varie metodologie di inserimento è utile ipotizzare come questi avranno impatto sulle prestazioni del albero. In particolare ci aspettiamo di vedere, in caso di un elevato numero di chiavi ripetute, una migliore prestazione nella ricerca, dovuta alla ridotta altezza del albero, utilizzando il metodo di inserimento basato sulla lista concatenata e allo stesso modo un altezza inferiore anche per il metodo con flag booleana che dovrebbe rendere piu bilanciato l'albero e quindi ridurne l'altezza. Al contrario, il maggior numero di operazioni necessare dovrebbe rendere il metodo con lista il più lento nel inserimento di nuovi dati.

	Complessità al caso peggiore	Complessità al caso medio
Spazio	$\Theta(n)$	$\Theta(n)$
Inserimento	$O(n)$	$O(h)$
Ricerca	$O(n)$	$O(h)$
Cancellazione	$O(n)$	$O(h)$

Figura 2: Complessità degli algoritmi di ABR

Il nostro obiettivo in questo test è verificare sperimentalmente la veridicità delle varie complessità descritte nella figura 2 e capire sotto quali condizioni un albero è più conveniente di un altro, confrontandoli, a parità di numero di chiavi, in base al tempo reale che impiegano ad eseguire le operazioni.

3 Documentazione del codice

3.1 Schema del contenuto e interazione tra i moduli

Per svolgere i test ho dovuto prima di tutto implementare il codice per creare la nostra struttura dati, questo avviene al interno del modulo **BST.py**. In questo modulo ho definito una classe **Tree** il cui builder prende come input un valore che sarà la chiave della radice e la passa al costruttore della classe **Node** che crea il primo nodo del albero. La classe **Node** definisce ogni singolo nodo del albero e il suo costruttore inizializza la variabile **key** al valore che gli è stato fornito. Questa classe, oltre agli ovvi attributi key, left (figlio sinistro), right (figlio destro) e p (padre) contiene l'attributo **b** (boolean) per l'implementazione del inserimento con flag booleana e l'attributo **duplicate** che rappresenta il puntatore al nodo duplicato successivo al interno della lista concatenata, in caso si utilizzi questa tecnica. Ho successivamente creato il modulo **dataset.py** con lo scopo di generare i dati su cui eseguire i test. In fine ho creato il modulo **benchmark.py** dove troviamo i metodi che svolgono i test.

3.2 Descrizione dei metodi implementati

Vediamo ora i metodi e le funzioni delle classi e dei moduli di cui finora abbiamo parlato.

- **Tree**
  - **normalInsert(z)** : esegue l'inserimento di un nodo con chiave z con il metodo classico.
  - **booleanInsert(z)** : esegue l'inserimento di un nodo con chiave z con il metodo della flag booleana.
  - **listInsert(z)** : esegue l'inserimento di un nodo con chiave z con il metodo della lista concatenata.
  - **insert(z, insertType)** : esegue l'inserimento di un nodo con chiave z con il metodo indicato dalla variabile insertType.
  - **search(k)** :esegue la ricerca del primo nodo del albero con chiave k.
- **dataset.py**
  - **generate_random_dataset(tests, times, dim)** : genera un dataset completamente casuale.
  - **generate_random_high_repetitions_dataset(tests, times, dim)** : genera un dataset completamente casuale con alta frequenza di ripetizioni.
  - **generate_unbalanced_tree_dataset(tests, terms, dim)** : genera un dataset completamente casuale con alta frequenza di ripetizioni e poi lo riordina, ottenendo in questo modo un dataset che generi un albero completamente sbilanciato.
  - Spiego meglio la struttura di questi dataset nella sezione 4.1.
- **benchmark.py**
  - **createTree(dataset, insertionType)** : utilizzando il metodo insert della classe Tree restituisce un albero creato inserendo le chiavi contenute nel array dataset con il metodo indicato da insertionType.
  - **benchmark(dataset, insertionType)** : la funzione crea l'albero con la funzione createTree, procede poi cronometrando il tempo di inserimento del ultimo elemento del dataset e infine cronometra il tempo di ricerca di un elemento scelto casualmente dal array. Per ogni test viene scartato il tempo più alto per ridurre l'influenza di fattori esterni sui risultati e poi sui valori restanti viene calcolata una media. Le misurazioni verrano comunque spiegate più in dettaglio nella sezione 4.2

4 Descrizione degli esperimenti condotti e analisi dei risultati sperimentali

4.1 Dati utilizzati

Per gli esperimenti che andremo ad eseguire utilizzeremo diversi tipi di dataset per poter coprire il maggior numero possibile di casi che si potrebbero incontrare nel effettivo utilizzo di questi algoritmi. Un dataset di test è costituito da un insieme di "sotto-dataset" che rendono i test completi. Il dataset è quindi composto da 3 tipi di array:

- Random Dataset (RD) ovvero dataset di interi generati casualmente con numeri tra 0 e la dimensione del array
- High Repetitions Dataset (HRD) ovvero dataset di interi generati casualmente con numeri tra 0 e la metà della dimensione del array, in modo tale da aumentare il numero di ripetizioni
- Worst Case Dataset (WCD) ovvero un HRD ordinato in modo crescente, in modo tale da ottenere un albero completamente sbilanciato

Ognuno di questi array viene generato in diverse dimensioni (100, 200, 300,...,1000 elementi) in modo da poter vedere la complessità delle operazioni al crescere del dataset. Infine ognuno di questi test viene eseguito 500 volte, per poter minimizzare l'errore dovuto a fattori esterni al programma calcolando la media del tempo impiegato per ogni test.

4.2 Misurazioni

La funzione di benchmark riceve in input il dataset su cui deve eseguire il test e il metodo di inserimento che dovrà testare. Per ogni array nel dataset costruisce quindi un albero con il metodo di inserimento specificato tralasciando l'ultimo elemento. Procederà poi a cronometrare il tempo impiegato per l'inserimento del ultimo elemento simulando perciò il tempo impiegato ad inserire un dato in un albero già costruito. In seguito cronometra la ricerca di un intero scelto casualmente al intero del array. Questi test vengono eseguiti come detto in precedenza 500 volte per ogni dimensione e viene calcolata una media per ogni dimensione ottenendo come output due array: un array contenente la media dei tempi di inserimento per ogni dimensione e uno contenente la media dei tempi di ricerca per ogni dimensione.

4.3 Risultati sperimentali e commenti analitici

4.3.1 Inserimento

Iniziamo subito notando come i risultati del test che vediamo in figura 3 confermino quanto ipotizzato dalla nostra analisi teorica nella sezione 2.3, ottenendo $\Theta(h)$ come andamento delle operazioni di inserimento su dataset RD e HRD, che rappresentano il caso medio. Andando più in dettaglio con l'analisi possiamo notare come i casi medi, indifferentemente dal tipo di dataset, abbiano un andamento molto simile a parità di algoritmo utilizzato per l'inserimento, coronando l'algoritmo classico come il più rapido, seguito poi dalla lista concatenata e infine dalla flag booleana. A differenza di quanto ipotizzabile, data la maggiore complessità delle operazioni per la creazione della lista concatenata, l'algoritmo con lista risulta più prestante di quello booleano probabilmente aiutato dal fatto che incorporando i nodi uguali in uno solo riduce l'altezza del albero e quindi il numero di cicli necessari per l'inserimento, a differenza della flag booleana che si limita a cercare di bilanciare meglio l'albero, mantenendo però il numero di nodi invariato.

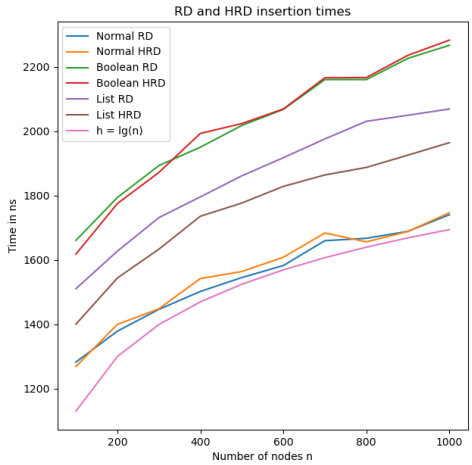

Figura 3: Inserimento su RD e HRD

Anche per quanto riguarda i risultati mostrati in figura 4 possiamo notare come gli esperimenti abbiano confermato la nostra analisi teorica mostrando un andamento $\Theta(n)$ per tutti gli algoritmi nel caso peggiore. Non ci sorprende vedere come l'algoritmo basato sulla lista concatenata risulti il più rapido, in quanto riducendo il numero di nodi riduce anche l'altezza del albero (che nel caso peggiore risulta essere una lista) permettendo cosi di ottenere i risultati migliori. Risulta interessante però notare come l'algoritmo classico risulti più prestante di quello con flag booleana.

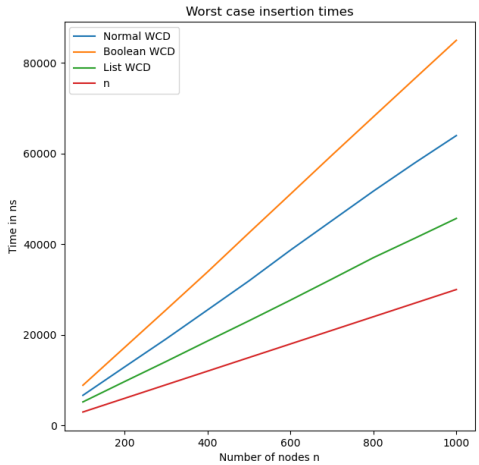

Figura 4: Ricerca

4.3.2 Ricerca

Per quanto riguarda le prestazioni della ricerca mostrate in figura 5 possiamo notare come i risultati sperimentali abbiano confermato le nostre ipotesi con un andamento $\Theta(h)$ per i dataset RD e HRD. Salta subito al occhio come tutti gli algoritmi abbiano prestazioni migliori nel caso di dataset ad alte ripetizioni, dato dal fatto che sono stati ottimizzati appositamente per questa evenienza e non ci sorprende vedere come sia su dataset RD che HRD l'algoritmo con lista concatenata risulti il più prestante. Aggiungiamo inoltre che su dataset RD le prestazioni sono quasi identiche per tutti gli algoritmi con un leggero miglioramento per quanto riguarda l'algoritmo a lista concatenata mentre per i dataset HRD l'algoritmo a lista concatenata risulta decisamente più prestante rispetto agli altri due che danno risultati molto simili.

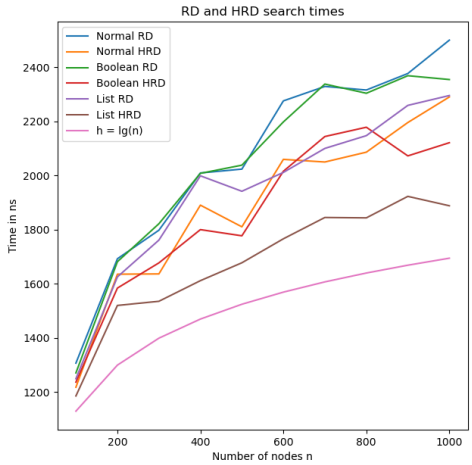

Figura 5: Ricerca su RD e HRD

Come per i casi precedenti anche su dataset WCD i risultati in figura 6 sono concordanti con la nostra previsione di una complessità $\Theta(n)$. Nuovamente l'algoritmo con lista concatenata risulta nettamente più prestante degli altri anche se vediamo un leggero miglioramento di prestazioni da parte del algoritmo con flag booleana rispetto a quello con implementazione classica.

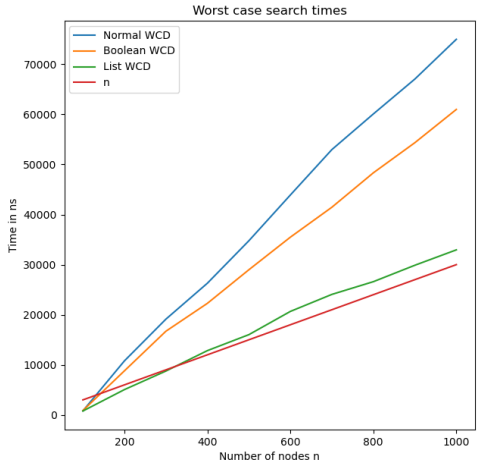

Figura 6: Ricerca su WCD

4.4 Tesi e sintesi finale

Riassumiamo adesso cosa possiamo estrapolare dai nostri test:

- Entrambi i metodi analizzati hanno complessità $\Theta(h)$ nel caso medio come avevamo previsto nell'analisi teorica.
- I due metodi nel caso peggiore hanno complessità $\Theta(n)$ confermando l'analisi teorica.
- In caso sia critica la necessità di creare rapidamente l'albero binario di ricerca, anche a costo della perdita di prestazioni in tutte le altre operazioni, il metodo di inserimento classico é il più indicato.
- Se i tempi di creazione sono meno importanti il metodo di inserimento con lista concatenata risulta il più prestante.
- Il metodo con flag booleana non risulta portare nessun vantaggio se non nella ricerca nel caso peggiore. La possibilità di randomizzare l'input riducendo l'evenienza di questo caso rende futile l'utilizzo di questo algoritmo considerando la perdita di prestazioni in ogni altro caso.

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.