Ingegneria degli Algoritmi

A.A. 2017-2018

Prova Pratica - Prima Prova in Itinere

Problema 1:

Binary search tree con lazy deletion

Briscese Filippo Maria

0228612

Descrizione dell'algoritmo

Ho implementato le richieste del problema cercando di estendere il codice fornito, in particolare creando due nuove sottoclassi: `LazyNode` e `DictLazyTree`.

`LazyNode`: è semplicemente un `BinaryNode` (classe nei codici forniti) con un attributo in più, `deleted`; questo attributo è un `bool` che indica lo stato di "cancellazione" del nodo.

`DictLazyTree`: partendo da `DictBinaryTree` non ho aggiunto attributi ed ho definito alcune funzioni che estendono o sostituiscono i metodi originali. Qui di seguito sono descritti i metodi propri di questa sottoclasse:

- `delete(chiave k)`[1] `--> bool`: sfruttando `searchNode` di `DictBinaryTree` trova, se esiste, il nodo, altrimenti ritorna `None`; se il nodo trovato non è ancora segnato come cancellato, lo segna, altrimenti ritorna `None`.
- `insert(chiave k, valore v)`: Inserisce un nuovo nodo "pigro" esaminando singoli nodi dell'albero partendo dalla radice, controllando se il nodo in esame:
  1. è una foglia cancellata, allora sostituisce il vecchio nodo;
  2. è un nodo cancellato e se ci sono le condizioni (cioè se il massimo del sottoalbero sinistro è minore di k e se il minimo del sottoalbero destro è maggiore di k), allora sostituisce il vecchio nodo;
  3. infine, se è un nodo non cancellato oppure non rientra nei punti 1. e 2., confronta i suoi figli per stabilire in quale sottoalbero continuare prendendo in esame la radice del sottoalbero scelto.

  La funzione continua pari all' "originale" nel caso non si rientri mai nei tre punti sopra, inserendo il nuovo nodo come foglia nella posizione opportuna.
- `search(chiave k)`[1]: cerca il nodo con chiave k sfruttando `searchNode` di `DictBinaryTree`; ritorna il nodo trovato se il nodo esiste e non è segnato come cancellato, altrimenti se il nodo non esiste o è segnato cancellato ritorna `None`.

È presente un altro metodo proprio (non richiesto nella traccia del problema), `minKeySon(nodo)`; questo metodo non è altro che la versione speculare di `maxKeySon(node)`[2] che è possibile trovare tra i metodi di `DictBinaryTree`, e non fa altro che ritornare il nodo più a sinistra di tutti.

[1] : Se presenti nodi duplicati considererà sempre e solo quello più vicino alla radice. Naturalmente è possibile implementare una ricerca più "profonda".

[2] : Ho apportato delle piccolissime modifiche in questo metodo a causa di alcuni problemi con errori di tipo durante l'esecuzione. Per approfondire andare alla riga 31 di *dictBinaryTree.py*.

Analisi del tempo di esecuzione

Tempi Teorici

- `delete`
  In questo caso è probabile impiegare meno tempo rispetto al metodo in `DictBinaryTree`, in virtù del fatto che non è necessario "aggiustare" la struttura dell'albero alla cancellazione. Controlla il nodo che ritorna `searchNode` di `DictBinaryTree`, spendendo quindi $O(h) + O(1) = O(h)$.

- `insert`[3]
  Questo metodo è più lento di quello in `DictBinartTree`, perché per ogni nodo cancellato preso in esame è necessario conoscere il massimo del sottoalbero sinistro e il minimo del sottoalbero destro per stabilire se è possibile rimpiazzarlo. In particolare, se venisse passato, nel caso peggiore, un intero ramo alto h di nodi cancellati non rimpiazzabili la funzione impiegherebbe al massimo $h \cdot O(h) = O(h^2)$[4]; se invece volessimo inserire un nodo "attraversando" un ramo alto h di nodi non cancellati impiegherebbe solamente $O(h)$.

- `search`
  I tempi teorici non differiscono dal metodo originale visto che non fa altro che controllare quello che ritorna `searchNode` di `DictBinaryTree`. Verrà quindi speso $O(h)$ per `searchNode` (dove h è l'altezza dell'albero) e $O(1)$ per i controlli, risultando quindi $O(h)$.

Funzione	*Condizioni*	*Tempo medio*
`delete`	(250 "segnature" per esecuzione)	0,12758
`insert`	Albero senza nodi cancellati (1000 inserimenti per esecuzione)	0,09132
	Albero con parte dei nodi cancellati (195 inserimenti per esecuzione)	0,13624
	* Albero completamente cancellato (1000 inserimenti per esecuzione)	1,14080
`search`	Di elementi presenti nell'albero (1000 ricerche per esecuzione)	0,12781
	Di elementi non presenti nell'albero (2000 ricerche per esecuzione)	0,11801
	Di elementi "cancellati" nell'albero (250 ricerche per esecuzione)	0,12006

Tabella 1. *Tempi (in ms) medi calcolati su 1000 esecuzioni. (* : 5 esecuzioni invece che 1000)*

[3] : Si potrebbero ottenere tempi migliori se ad esempio ogni nodo portasse con sé le informazioni che si ottengono con `minKeySon` e `maxKeySon`.
[Tempo speso per questo singolo passo in `insert`: $O(1)$ vs $O(h-1)$; Tempo nuovo `insert`: $O(h)$]

[4] : $O(k-1)$ per un nodo ad altezza k per calcolare `maxKeySon` e `minKeySon`, maggiorando: $O(h) \cdot h$ nodi.

Tempi Sperimentali

Per testare il codice ho creato una lista (da qui in poi `lista`) di 1000 interi ϵ [0, 9999] utilizzando `random.randint`, poi ho creato una seconda lista (da qui in poi `listaP`) scegliendo 250 elementi della prima con `random.sample`. Ho raccolti i tempi eseguendo in quest'ordine:

- `insert` usando `lista` per creare un albero;
- una volta creato, `search` cercando ogni elemento di `lista`;
- `delete` cancellando dall'albero creato gli elementi di `listaP`;
- `search` cercando gli elementi di `listaP` (e quindi segnati cancellati nell'albero);
- `insert` di parte degli elementi cancellati (i primi 195 di `listaP`);
- `search` di elementi non presenti nell'albero.

Il tutto è stato ripetuto per 1000 esecuzioni.

Per testare il caso "`insert` in un albero completamente cancellato" ho creato un albero che a partire dal sottoalbero destro della radice è completo ad ogni suo livello, di altezza uguale all'albero usato nei test precedenti, $h = 20$; infine ho inserito ripetutamente un nuovo nodo con chiave maggiore della chiave massima nell'albero, cancellandolo appena non fosse finita l'operazione (ovviamente tenendo conto in questo passo del solo tempo speso da `insert` e non di quello speso da `delete`). Il tutto è stato ripetuto per 5 volte.

I risultati dei tempi medi possono essere letti in **Tabella 1**. Si può notare che i tempi di `delete` e `search` sono molto vicini tra loro, questo era prevedibile visto che si appoggiano alla stessa funzione `searchNode`; si può anche notare che `insert` è decisamente più veloce se si opera su un albero senza elementi cancellati. Non può non saltare all'occhio il tempo di insert nel caso peggiore, circa 10 volte più grande di qualsiasi altro test, risultato facilmente attendibile considerati i tempi teorici.

Un risultato che invece non mi aspettavo è che i tempi di `delete` e `search` fossero più vicini a quelli di `insert` su un albero con cancellazioni piuttosto che a quelli di `insert` su un albero senza cancellazioni; credo che questo dipenda dal fatto che l'albero con cancellazioni su cui si sta inserendo ha un *"tasso di cancellazione"*[5] alla prima chiamata del *25%*, sicuramente non abbastanza alto da "peggiorare" il tempo teorico di `insert` fino a $O(h^2)$, inoltre non è completo in tutti i livelli come l'albero costruito appositamente per il test del caso peggiore.

[5] : $Tasso\ di\ cancellazione = \frac{numero\ nodi\ cancellati}{numero\ nodi\ totali}$.

Uso della Lazy Deletion

Come ho potuto constatare dai test avere un albero con la maggior parte dei nodi segnati come cancellati risulta essere molto svantaggioso appena è necessario effettuare un inserimento.

Usare quindi la lazy deletion potrebbe essere vantaggioso nel caso si stesse operando su un albero di dimensioni molto grandi, inserendo e cancellando nodi in modo continuo e con un rapporto inserimenti/cancellazioni circa uguale ad 1. La lazy deletion fa in modo che ad ogni operazione non sia necessario aggiustare l'albero, nella speranza che gli inserimenti successivi "riempino i buchi" e quindi non vadano ad aumentare l'altezza dell'albero. Inoltre nel caso si avesse un albero ben bilanciato saremmo molto interessati nel mantenere questa sua struttura, e la lazy deletion ci aiuta in questo visto che ci permette di non eseguire tagli, rotazioni e qualsiasi aggiustamento di sorta alla struttura.

Nell'uso prolungato della lazy deletion si potrebbero avere una serie di inserimenti e cancellazioni che non siano "salutari" per l'albero, ovvero che creino dei rami "morti" (numero elevato di cancellazioni) ed ingrandiscano eccessivamente le sue dimensioni in altezza; sarà quindi utile effettuare una "potatura" (eliminazione reale dei nodi cancellati) dell'albero periodicamente per mantenere "fruttuoso" il suo utilizzo.

Quindi in conclusione, nell'uso di un albero binario di ricerca, se non siamo interessati ad eliminare nodi continuamente, anzi se stiamo apportando un equilibrato ricambio di nodi (cancellando ed inserendo), l'uso della lazy deletion dovrebbe darci la possibilità di risparmiare tempo evitandoci di eseguire operazioni di manutenzione dell'albero che non sono necessarie per questo utilizzo, anzi potrebbero incrementare i tempi senza portare nessun vantaggio.