Documentazione Progetto 2
Gruppo 6

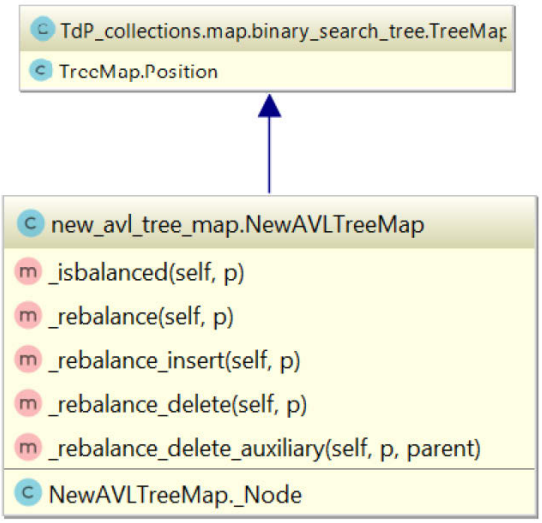
Antonino Durazzo

Valentino Vastola

Francesco de Pertis

1) Classe NewAVLTreeMap

UML-Diagram NewAVLTreeMap class



NewAVLTreeMap._Node

È stata ridefinita la sottoclasse Node di TreeMap nel cui __init__(element, parent=None, left=None, right=None) verranno inizializzati gli attributi nel nodo e l'attributo _balance_factor impostato a 0 di default.

NewAVLTreeMap._isbalanced(p)

:param: p: Positon

:return: True or False

Complessità: O(1)

Metodo che restituisce True se il nodo è bilanciato altrimenti False. Prima di verificare il valore del balance factor del nodo, viene effettuata una validazione della position in input.

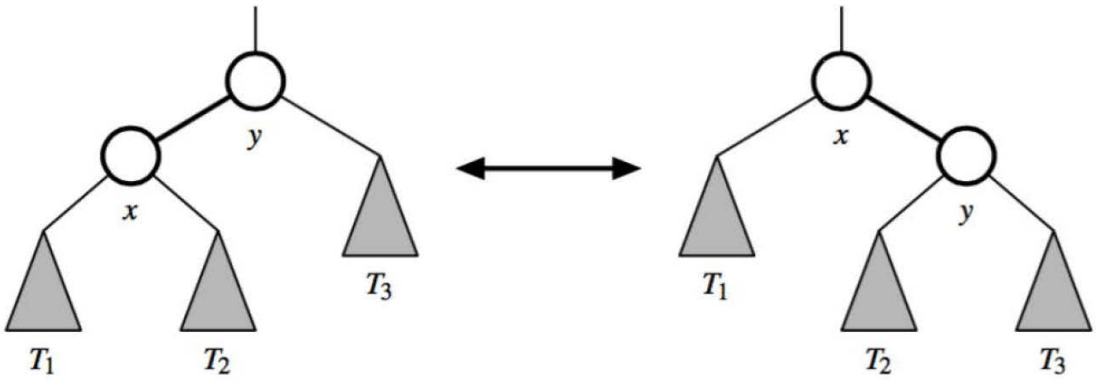
NewAVLTreeMap._rebalance(p)

:param: p: Positon

:return: padre del nodo nel caso in cui venga effettuata una singola rotazione

Complessità: O(1)

Metodo che effettua il ribilanciamento dei nodi dell'albero dopo aver effettuato l'opportuna rotazione. Tutte le 4 casistiche di rotazione possono essere ricondotte nell'applicazione di due rotazioni logiche, una rotazione a destra e una a sinistra.



Pertanto abbiamo analizzato come potrebbe cambiare il valore del balance factor dei nodi dopo aver effettuato una delle due rotazioni concludendo che si può mettere in relazione il valore del balance factor prima della rotazione con il valore che dovrà assumere:

OldB(i) = Balance Factor del nodo i prima della rotazione

NewB(i) = Balance Factor del nodo i dopo la rotazione

Caso rotazione a destra:

NewB(y) = OldB(y) + 1 - min(0,OldB(x))

NewB(x) = OldB(x) + 1 + max(0,NewB(y))

Caso rotazione a sinistra:

NewB(x) = OldB(x) - 1 - max(0,OldB(y))

NewB(y) = OldB(y) - 1 + min(0,NewB(x))

Dunque nel caso di una doppia rotazione, a seconda della posizione dei tre nodi coinvolti nella rotazione dovrò effettuare prima un allineamento dei nodi per poi effettuare una nuova rotazione per bilanciarli quindi, le formule sopra riportate verranno applicate dopo ogni rotazione.

NewAVLTreeMap._rebalance_insert(p)

:param: p: Positon

Complessità: O(log n)

È stata ridefinita la _rebalance_insert che ricorsivamente effettua l'aggiornamento dei valori del balance factor degli antenati finchè quel determinato sottoalbero risulta essere bilanciato ovvero, che il valore risultante del balance factor risulti diverso da 0 o che si sia arrivati alla radice.

Per aggiornare i valori del balance factor dei genitori, a ogni istanza della ricorsione è stato osservato che dopo l'inserimento di un determinato nodo nell'albero, questo renda sbilanciato più verso destra o più verso sinistra, di un fattore di 1, il sottoalbero di cui fa parte a seconda se esso sia stato inserito come figlio di sinistra o di destra.

Prima di aggiornare il valore del padre nell'istanza della ricorsione, viene controllato se il corrente nodo è bilanciato, se non lo è allora viene invocata la rebalance. La _rebalance_insert termina dopo

l’esecuzione della rebalance, in quanto effettuare l’aggiornamento degli antenati non serve perché l’albero risulta essere bilanciato e i valori dei balance factor di tutti i nodi risulteranno corretti.

NewAVLTreeMap._rebalance_delete_auxiliary(p)

:param: p: Positon

Complessità: O(log n)

Funzione che effettua ricorsivamente l’aggiornamento dei valori del balance factor degli antenati finché quel determinato sottoalbero risulta essere bilanciato, ovvero che il valore risultante del balance factor risulti 0 o che si sia arrivati alla radice.

Per aggiornare i valori del balance factor dei genitori, a ogni istanza della ricorsione è stato osservato che dopo l’inserimento di un determinato nodo nell’albero, questo renda sbilanciato più verso destra o più verso sinistra, di un fattore di 1, il sottoalbero di cui fa parte a seconda se esso sia stato inserito come figlio di sinistra o di destra.

Prima di effettuare una nuova ricorsione ove necessaria, viene controllato se il corrente nodo è bilanciato, se non lo è allora viene invocata la rebalance.

NewAVLTreeMap._rebalance_delete(p)

:param: p: Positon che rappresenta il padre del nodo eliminato

Complessità: O(log n)

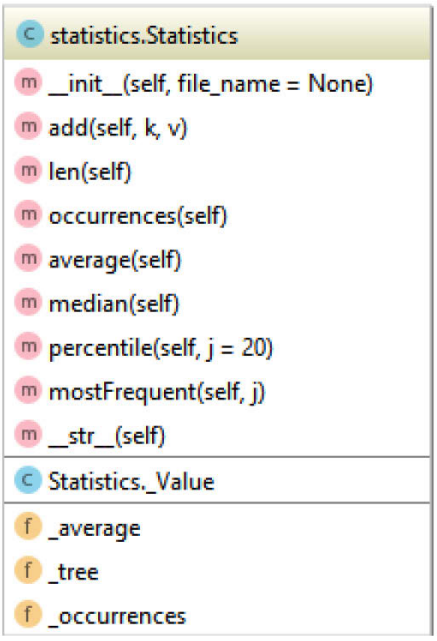
È stato ridefinito _rebalance_delete che si occupa di aggiornare i valori dei balance factor degli antenati invocando la _rebalance_delete_auxiliary se il balance factor del nodo in input risulti 0, inoltre aggiorna il saldo del balance factor del nodo in input e viene controllato se il balance factor del nodo dopo l’aggiornamento risulti bilanciato altrimenti viene invocata la rebalance.

Per aggiornare il balance factor del nodo in input vengono distinti 4 casi

1) Viene eliminato un nodo da un padre che aveva solo il nodo eliminato come figlio, nel caso il balance factor del padre diventa 0
2) Viene controllato se è stato eliminato il figlio sinistro, se si allora il saldo del balance factor del padre risulterà decrementato di un fattore di 1 perché il nodo sarà sbilanciato verso destra.
3) Viene controllato se è stato eliminato il figlio destro, se si allora il saldo del balance factor del padre risulterà incrementato di un fattore di 1 perché il nodo sarà sbilanciato verso sinistra.
4) Viene eliminato un nodo il cui padre risulterà avere due figli, in questo caso si deve capire da dove sia stato eliminato il nodo per poter aggiornare il valore del balance factor e lo si fa guardando il numero dei figli del figlio destro e sinistro del nodo in input.

2) Classe Statistics

UML-Diagram NewAVLTreeMap class



Statistics._Value

È stata definita come sottoclasse privata per poter assegnare al value della mappa contenuta in NewAVLTreeMap una struttura dati composta da *_frequency* e *_total*.

Statistics.__init__(file_name = None)

:param: file_name: path del file da importare durante l'inizializzazione della classe

Complessità: O(n*log n)

Inizializza un oggetto Statistics. I suoi attribute sono *_tree*, un NewAVLTreeMap, *_occurrences*, un contatore per le occorrenze totali nel NewAVLTreeMap e *_average*, una variabile contenente la media al fine di migliorare le complessità computazionali delle funzioni Statistics.occurences() e Statistics.average(). Se il parametro *file_name* non è nullo, importa il file e passa le varie coppie al metodo Statistics.add(k, v).

Statistics.add(k, v)

:param: k: key da inserire o aggiornare nell'albero.

:param: v: value da inserire o sommare nell'albero

Complessità: O(log n)

Controlla se la key passata come parametro è presente nell'albero. Se non è presente la aggiunge, setta la sua *frequency* a 1 e il suo totale a v, altrimenti incrementa la sua *frequency* associata di 1 e il suo *total* di v. Incrementa l'attributo *_occurences* di uno e aggiorna l'attributo *_average*.

Statistics.len()

Complessità: O(1)

Restituisce il numero di chiavi presenti nella mappa.

Statistics.occurrences()

Complessità: O(1)

Restituisce la somma delle frequenze di tutti gli elementi presenti nella mappa.

Statistics.average()

Complessità: O(1)

Restituisce la media dei valori di tutte le occorrenze presenti nel dataset.

Statistics.median()

Complessità: O(n*log n)

Restituisce la mediana delle key presenti nel dataset invocando il metodo Statistics.percentile(50).

Statistics.percentile(j)

Complessità: O(n*log n)

Restituisce il *j*-imo percentile, per 1<= *j* <= 99 delle lunghezze delle key. Lancia un'eccezione se *j* non è compreso nel range. Definisce l'indice della key che si sta cercando, poi itera n volte a partire dal primo elemento dell'albero.

Statistics.mostFrequent(j)

Complessità: O(n*log n)

Restituisce la lista delle j key più frequenti. Crea una lista contenente le position dei nodi dell'albero, le ordina in maniera decrescente in base alla frequenza tramite l'ausilio della funzione *_mergesort*, dopodiché converte le prime *j* position nelle relative key per restituire infine i primi *j* elementi della lista.

3) find_repetition(dir)

:param: dir: path della directory che contiene i file da esaminare

:return: lista dei file replicati

Complessità: O(n) dove n è il numero di file da esaminare

Questa funzione prende in input una directory e ritorna la lista dei file replicati. È stata utilizzata una la funzione hash di default di Python per trasformare il contenuto dei file analizzati in interi di 64 bit. Tali interi sono tati poi utilizzati come indici in una ProbeHashMap. Tale mappa associa gli hashcode a una lista dei nomi dei file con medesimo contenuto. Dopodiché viene effettuato uno scorrimento di tale HashMap al fine di individuare le liste con dimensione maggiore di uno e quindi i file che hanno una o più copie.
La scelta di tale implementazione è mirata a trovare il giusto compromesso tra complessità computazionale e probabilità di collisione.

4) circular_substring(P, T)

:param: P: pattern

:param: T: testo

:return: True or False se il pattern è stato trovato all'interno del testo

Complessità: O(n+m) dove n è la lunghezza del testo e m e la lunghezza del pattern

Questa funzione rappresenta una rivisitazione dell'algoritmo di Knuth-Morris-Pratt e opera allo stesso modo, ma si estende il confronto del testo con il pattern anche se si sia arrivati alla fine del testo. Dopo aver controllato l'esistenza di sottostringhe all'interno del testo, continua a confrontare i caratteri del pattern con l'inizio del testo, fino ad arrivare al più a m-k caratteri degli n del testo. Qualora non ci sia stata una discrepanza quando si è arrivati alla fine del testo, ci si aspetta di trovare il pattern nel testo a partire dal carattere fino a quel punto matchato con i primi caratteri del testo e non oltre.

Quando si deve ricominciare dall'inizio del testo per trovare un pattern circolare, allora la funzione di fallimento non dovrà essere richiamata perché nel caso di una discrepanza successiva l'algoritmo deve terminare.