Esercizio 1: La classe NewAVLTreeMap, da specifiche, deve fornire la stessa interfaccia di AVLTreeMap e memorizzare nei nodi i fattori di bilanciamento invece che le altezze dei sottoalberi. Per ottenere ciò, si è ridefinita la classe innestata \_Node(TreeMap.\_Node) per includere l’attributo \_balance\_factor e la classe NewAVLTreeMap estende TreeMap, ereditandone l’interfaccia pubblica.

Metodi della classe:

* \_isbalanced(self, p) -> bool: valuta se un nodo contenuto in una position è bilanciato. Il bilanciamento è definito come la differenza tra l’altezza del sottoalbero sinistro e l’albero destro, motivo per cui questo metodo valuta se il fattore di bilanciamento della posizione è nell’intervallo [-1,1]. Complessità computazionale O(1).
* def \_tall\_child(self, p, favorleft=False) -> (Position, bool): cerca il nodo figlio più alto della position p e lo restituisce. In più, restituisce un parametro booleano ad indicare se il nodo è figlio sinistro o destro. Complessità computazionale O(1).
* def \_tall\_grandchild(self, p) -> (Position, int): cerca il nodo nipote più alto a partire dalla position p e lo restituisce. In più, restituisce un intero indicante il tipo di rotazione che viene effettuata nella restructure in base alla configurazione iniziale dei nodi che determina la condizione di sbilanciamento, ossia RR -> singola rotazione a sinistra -> tipo 0, LL-> singola rotazione a destra -> tipo 1, RL e LR -> doppia rotazione -> tipo 2. Complessità computazionale O(1).
* def \_rebalance(self, p, insert): effettua il ribilanciamento dell’albero radicato nella position p nei casi di insert e delete, indicati dal parametro booleano di ingresso insert. Il ribilanciamento opera valutando e ricomputando propriamente il fattore di bilanciamento del padre della position p in ingresso, cioè a dire aggiornare di 1 il fattore di bilanciamento del padre in base all’operazione che ha scatenato la chiamata alla rebalance e alla relazione padre-figlio tra i nodi coinvolti. Se il nodo non è bilanciato, il metodo effettua la restructure e rivaluta i fattori di bilanciamento risalendo verso l’alto, controllando le condizioni di uscita. Complessità computazionale O(logn), con n numero di nodi dell’albero.
* \_rebalance\_insert(self, p): hook method che richiama la rebalance passando come parametro insert=True. Complessità computazionale O(logn), con n numero di nodi dell’albero siccome il metodo include operazioni a tempo costante e una chiamata alla \_rebalance().
* \_rebalance\_delete(self, p): hook method che richiama la rebalance passando come parametro insert=True. Per mantenere la coerenza con le operazioni inerenti la rebalance comune tra insert e delete, alcuni controlli devono essere effettuati sulla position in input alla \_rebalance\_delete(self, p) per aggiornare correttamente i fattori di bilanciamento. Complessità computazionale O(logn), con n numero di nodi dell’albero siccome il metodo include operazioni a tempo costante e una chiamata alla \_rebalance()
* \_update\_balance\_factor\_delete(self, p): metodo di utility per l’aggiornamento dei fattori di bilanciamento della position p di ingresso utilizzato dalla \_rebalance\_delete(). Complessità computazionale O(1).
* \_recompute\_balance\_factor(self, p, bf\_grandchild, rotation\_type): aggiorna i fattori di bilanciamento dei nodi coinvolti nella ristrutturazione intorno al nodo p tenendo conto del tipo di rotazione. Per fare ciò, si utilizzano il parametro bf\_grandchild e rotation\_type, che indicano, rispettivamente, il valore del fattore di bilanciamento del tall\_grandchild prima della ristrutturazione e il tipo di rotazione secondo la semantica adottata. Complessità computazionale O(1).
* \_change\_balance\_factor(self, p, v): metodo di utility per aggiornare il fattore di bilanciamento di p al valore v. Complessità computazionale O(1).
* \_retrieve\_balance\_factor(self, p): metodo di utility per accedere al campo protected \_balance\_factor del nodo. Complessità computazionale O(1).

Esercizio 2: Classe Statistics

La classe elabora statistiche su un dataset di key-value usando un NewAVLTreeMap i cui nodi contengono gli elementi key, frequency e total che rappresentano rispettivamente la chiave dell’elemento, la frequenza con la quale si presenta nel dataset e la somma dei valori per ogni elemento contenuto.

Inoltre, la classe presenta gli attributi occur e total che permetto di memorizzare il numero di occorrenze e la somma dei valori degli elementi nell’AVL permettendo così di velocizzare le operazioni di occurrences() e average().

La complessità del costrutto è O(nlogk) dove n rappresenta il numero di righe nel dataset mentre k il numero di chiavi.

Metodi della classe:

* add(k,v): aggiunge la coppia (k,v) nella mappa. Se la chiave k è già presente all’interno della mappa si aggiornano i campi frequency e total associati a quel nodo. La complessità è O(logk) dove k è il numero di chiavi all’interno del dataset;
* len(): restituisce il numero di elementi della mappa. Complessità O(1);
* occurrences(): restituisce il numero di occorrenze degli elementi della mappa. Complessità O(1);
* average(): restituisce il valore medio degli elementi inseriti nella mappa. Complessità O(1);
* percentile(j): calcola il j-esimo percentile, per j = 0, …, 99 delle frequenze delle chiavi definito come la chiave k tale che il j% delle occorrenze nel dataset abbia la chiave minore o uguale a k. Complessità O(k) dove k è il numero di chiavi all’interno del dataset;
* median(): richiamando il metodo percentile, restituisce il j = 50 percentile con complessità O(k);
* most\_frequences(): restituisce una lista contenente le j chiavi più frequenti. Per l’implementazione di questo metodo è stata utilizzata unaHeapPriorityQueue in quanto consente l’inserimento e la cancellazione in O(logj) con j numero di elementi nell’heap. Complessità O(klogj).

Esercizio 3: find\_repetition()

Funzione utilizzata per individuare file duplicati (file con nomi diversi ma con medesimi contenuti) all’interno di una cartella. Il suo funzionamento è basato sull’utilizzo di un dizionario e della funzione hash() built-in di python. Per ogni file, viene calcolato l’hash del contenuto utilizzato come chiave in una struttura associativa chiave-valore della forma (hash calcolato, lista dei file collidenti). Se i file hanno stesso contenuto il loro hash sarà lo stesso causando una collisione che comporterà un’aggiunta alla lista dei file collidenti con quel contenuto.

La complessità di tale funzione è lineare rispetto al numero di file presente nella cartella.