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(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);
- ***_tall_child(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);
- ***_tall_grandchild(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 ristrutturazione 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);
- ***_rebalance(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(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 una chiamata alla *_rebalance()*;
- ***_rebalance_delete(p)***: hook method che richiama la rebalance passando come parametro insert=False. 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(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(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(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(p, v)***: metodo di utility per aggiornare il fattore di bilanciamento di p al valore v. Complessità computazionale O(1);
- ***_retrieve_balance_factor(p)***: metodo di utility per accedere al campo protected *_balance_factor* del nodo. Complessità computazionale O(1).

Nella tabella seguente sono riassunte le complessità computazionali per ognuno dei precedenti metodi.

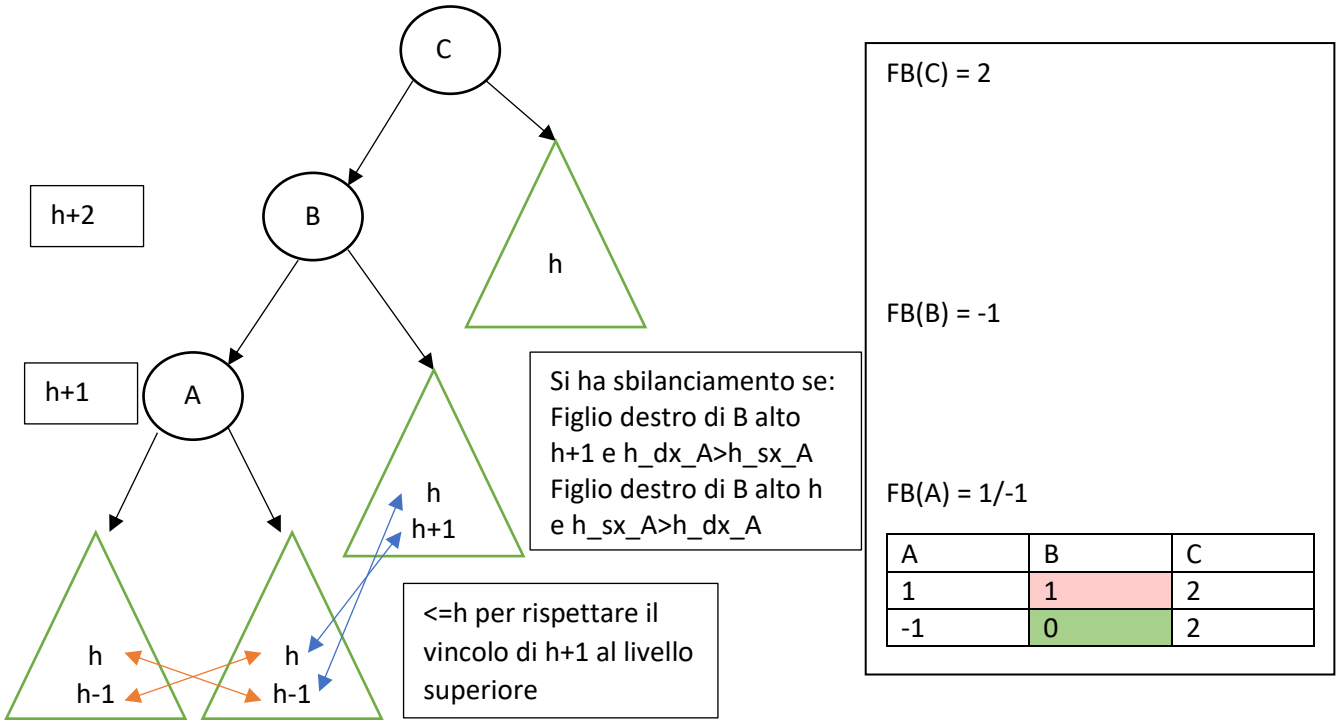
Metodo	Complessità
_isbalanced(p)	O(1)
_tall_child(p, favorleft=False)	O(1)
_tall_grandchild(p)	O(1)
_rebalance(p, insert)	O(log(n))
_rebalance_insert(p)	O(log(n))
_rebalance_delete(p)	O(log(n))
_update_balance_factor_delete(p)	O(1)
_recompute_balance_factor(p, bf_grandchild, rotation_type)	O(1)
_change_balance_factor(p, v)	O(1)
_retrieve_balance_factor(p)	O(1)

Calcolo dei fattori di bilanciamento

Per aggiornare correttamente i fattori di bilanciamento in seguito ai vari tipi di rotazione (dovuti a diversi casi di sbilanciamento) sono stati effettuate le considerazioni spiegate nel seguito.

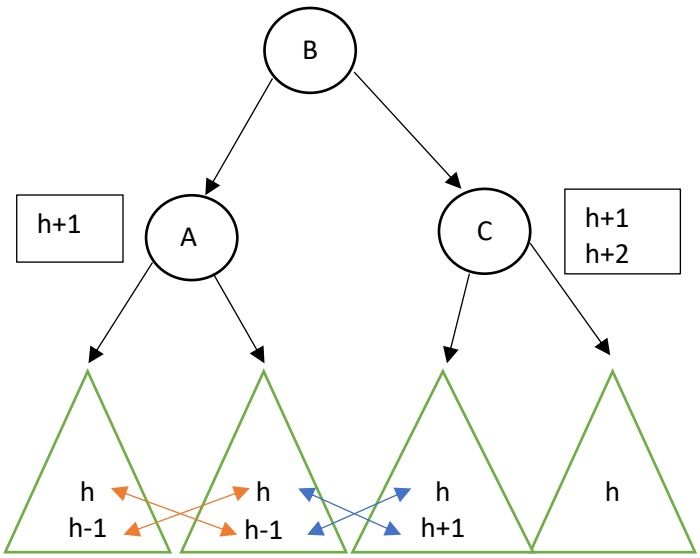
Left Left Imbalance Case (requires a single right rotation)

Al momento dello sbilanciamento la situazione è la seguente:



A	B	C
1	1	2
-1	0	2

Essendo il caso LL risolto tramite rotazione singola a destra, i sottoalberi radicati in A non subiscono variazioni e quindi non varia il fattore di bilanciamento di A, mentre bisogna aggiornare i fattori di bilanciamento di B e C in base al vecchio valore del fattore di bilanciamento di B.



FB(B) = 0/-1

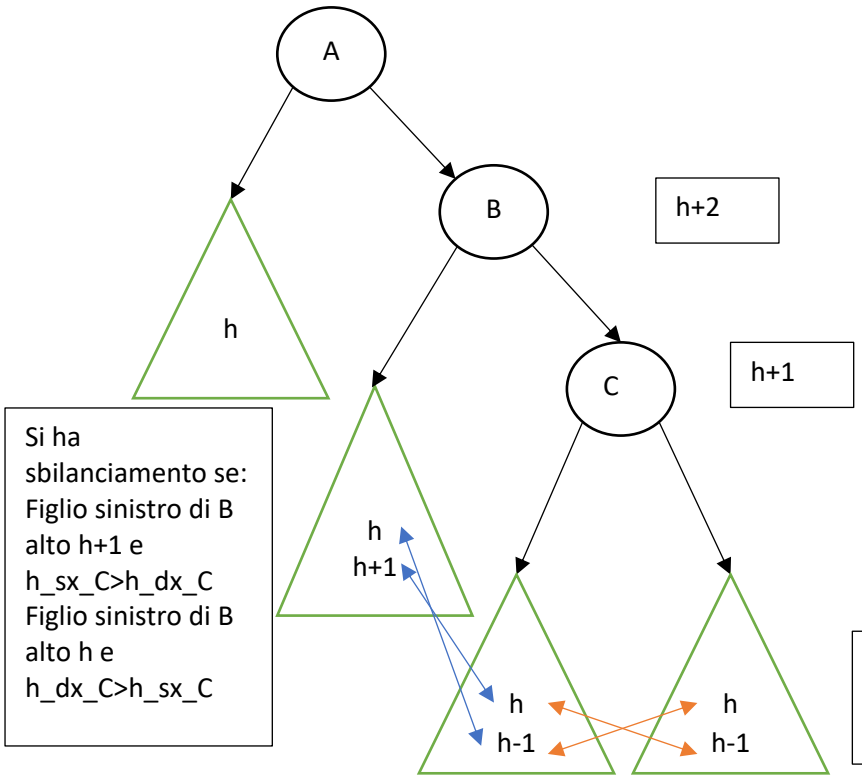
FB(A) = 1/-1

FB(C) = 0/1

A	B	C
1	0	0
-1	-1	1

Right Right Imbalance Case (requires a single left rotation)

Al momento dello sbilanciamento la situazione è la seguente:



Si ha sbilanciamento se:
Figlio sinistro di B alto h+1 e h_sx_C>h_dx_C
Figlio sinistro di B alto h e h_dx_C>h_sx_C

FB(A) = -2

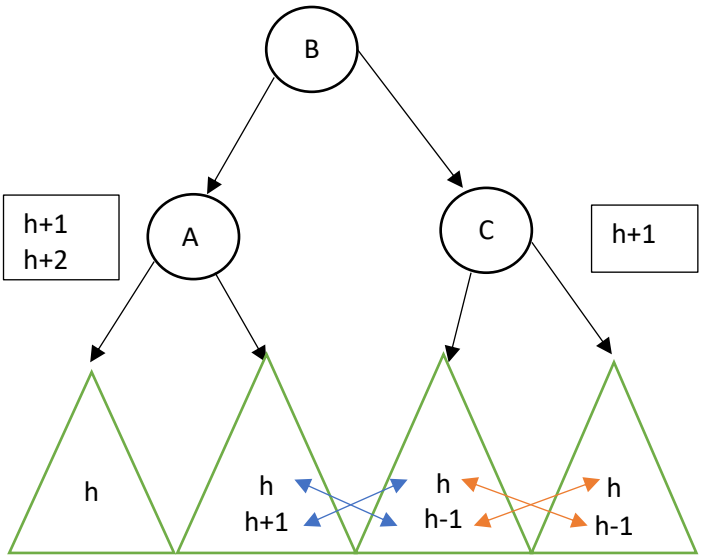
FB(B) = -1

FB(C) = 1/-1

A	B	C
-2	1	-1
-2	0	1

<=h per rispettare il vincolo di h+1 al livello superiore

Essendo il caso RR risolto tramite rotazione singola a sinistra, i sottoalberi radicati in C non subiscono variazioni e quindi non varia il fattore di bilanciamento di C, mentre bisogna aggiornare i fattori di bilanciamento di B e A in base al vecchio valore del fattore di bilanciamento di B, in modo specchiato rispetto al caso LL.



FB(B) = 1/0

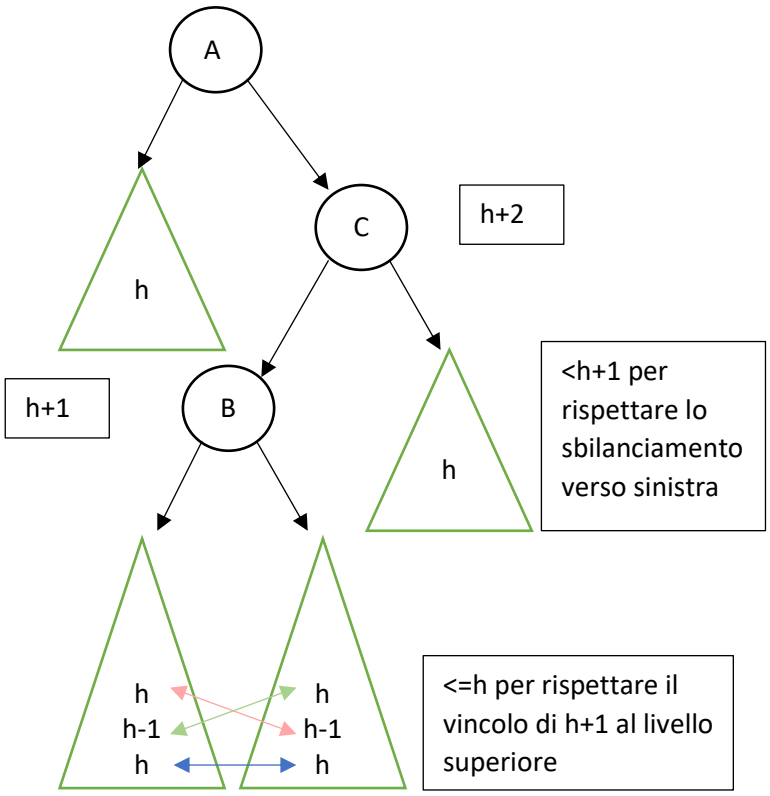
FB(A) = 1/-1

FB(C) = 0/1

A	B	C
0	0	-1
-1	1	1

Right Left Imbalance Case (requires a right rotation and a left rotation)

Al momento dello sbilanciamento la situazione è la seguente:



<h+1 per rispettare lo sbilanciamento verso sinistra

<=h per rispettare il vincolo di h+1 al livello superiore

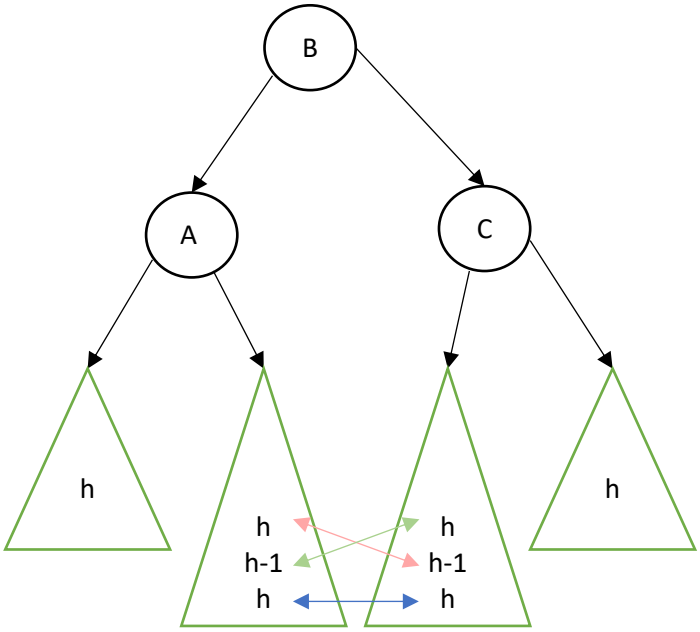
FB(A) = -2

FB(C) = 1

FB(B) = 1/-1

A	B	C
-2	1	1
-2	-1	1
-2	0	1

Essendo un caso a doppia rotazione, il fattore di bilanciamento del nodo attorno a cui avviene la rotazione è sempre 0, per cui in base al vecchio valore del fattore di bilanciamento di B andiamo ad aggiornare quelli di A e C.



FB(B) = 0

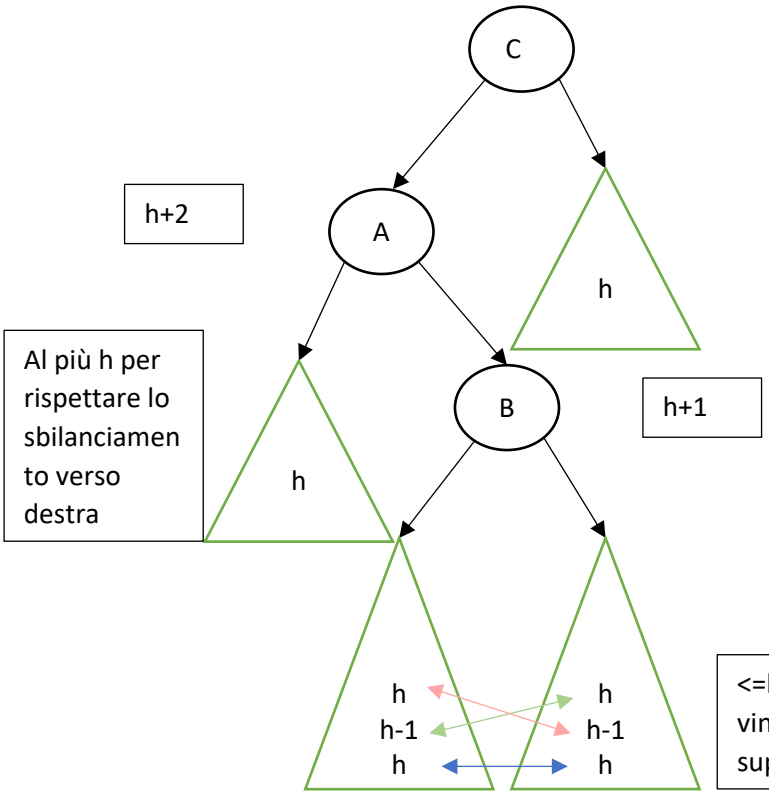
FB(A) = 0/1

FB(C) = -1/0

A	B	C
0	0	-1
1	0	0
0	0	0

Left Right Imbalance Case (requires a left rotation and a right rotation)

Al momento dello sbilanciamento la situazione è la seguente:



FB(C) = 2

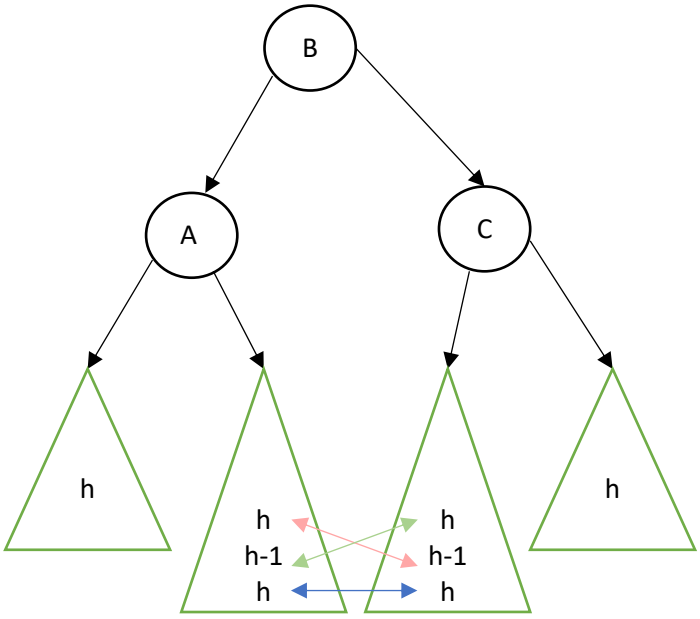
FB(A) = -1

FB(B) = 1/-1

A	B	C
-1	1	2
-1	-1	2
-1	0	2

<=h per rispettare il vincolo di h+1 al livello superiore

Si osserva come giungiamo agli stessi risultati del caso RL e quindi possiamo trattare gli aggiornamenti dei fattori di bilanciamento allo stesso modo per RL e LR.



FB(B) = 0

FB(A) = 0/1

FB(C) = -1/0

A	B	C
0	0	-1
1	0	0
0	0	0

Esercizio 2:

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. Per ogni nodo della mappa la key è la chiave di accesso al nodo mentre il valore è nella forma di una lista contenente frequency e total.

Inoltre, la classe memorizza negli attributi *occur* e *total* il numero di occorrenze e la somma dei valori degli elementi nella mappa, permettendo così di velocizzare le operazioni di *occurrences()* e *average()*.

La complessità del costruttore è O(nlogk) dove n rappresenta il numero di righe nel dataset da leggere, 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 distinte 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=20)***: 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. Il metodo itera l'AVL sui nodi, accedendo così ai campi chiave e frequenza, e confronta ad ogni iterazione la posizione del j-esimo percentile con la somma delle frequenze. Quando quest'ultima è maggiore e/o uguale alla posizione del percentile, viene restituita la chiave relativa all'iterazione corrente. 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);

- ***mostFrequent(j)***: restituisce una lista contenente le j chiavi più frequenti. Per l'implementazione di questo metodo è stata utilizzata una *HeapPriorityQueue* in quanto consente l'inserimento e la cancellazione in O(logj) con j numero di elementi nell'heap. Il metodo mantiene una coda di lunghezza j avente come chiavi la frequenza delle occorrenze e come valore le chiavi stesse. Per ogni nodo interno dell'AVL, viene inserita una corrispondenza (frequenza, chiave) all'interno della coda fin quando quest'ultima non risulta piena (possiede j elementi al suo interno). Per i restanti len(avl)-j elementi, si confronta la frequenza dell'ultimo elemento della coda con la frequenza del nodo corrente estratto dall'AVL. Se il primo valore è minore del secondo, viene estratto il minimo dalla coda e viene inserita una nuova corrispondenza (frequenza, chiave), relativa al nodo analizzato, nella coda. Alla fine di questa operazione avremo all'interno della coda i j elementi con frequenza più elevata. Complessità O(klog(j)) +j) (ciclo di ricerca dei most frequent e ciclo per creare il risultato), approssimabile a O(klog(j)) se k >> j.

Nella tabella seguente sono riassunte le complessità computazionali per ognuno dei precedenti metodi.

Metodo	Complessità
add(k,v)	O(log(k))
len()	O(1)
occurrences()	O(1)
average()	O(1)
percentile(j=20)	O(k)
median()	O(k)
mostFrequent(j)	O(klog(j))

Esercizio 3:

find_repetition() è una 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 dizionari e della funzione *hash()* built-in di Python. Per ogni file viene calcolato l'hash del suo contenuto e utilizzato come chiave in una struttura associativa chiave-valore della forma <hash calcolato, lista dei file collidenti>. Se due file hanno lo stesso contenuto, il loro hash con elevata probabilità, a patto della scelta di una funzione di hash sufficientemente resistente alle collisioni, sarà lo stesso, implicando che i file sono equivalenti. In tale caso l'identificativo del file collidente viene aggiunto alla lista di collisione opportuna.

La complessità di tale funzione è lineare rispetto al numero di file presente nella cartella se si considera il calcolo dell'hash a tempo costante O(1).

Per l'hashing sono a disposizione molte scelte. Il trade-off è sulla loro velocità di esecuzione e sullo spazio di memorizzazione dell'hash prodotto. Dato che la problematica non riguarda questioni di sicurezza si è ritenuto superfluo scomodare funzioni messe a disposizione dalle librerie crittografiche, che presentano tempi di calcolo potenzialmente insoddisfacenti e valori di hash di dimensione considerevole. Si è allora adottata come soluzione la semplice e veloce funzione built-in *hash()* che produce un valore intero su 64 o 32 bit attraverso semplici operazioni matematiche di somma e prodotto.

Esercizio 4:

Il problema della definizione di una sottostringa circolare può essere affrontato applicando l'algoritmo KMP, che garantisce complessità O(n+m), dove n e m sono le lunghezze del testo analizzato e del pattern da ricercare, con una piccola manipolazione del testo iniziale. Essa consiste nel concatenare al testo come suffisso i suoi primi m caratteri, simulando la circolarità del pattern. Così facendo, un'esecuzione

dell’algoritmo KMP sul nuovo testo preserva un tempo lineare, in particolare O((n+m)+m), avendo incrementato la dimensione del testo. Ma, dato che si assume nei problemi di pattern matching che n >> m, asintoticamente la complessità della soluzione può essere valutata come di O(n+m). Oppure, sotto un’altra ottica, la complessità totale è ordine di O(n+2m), in cui la costante moltiplicativa 2 può essere trascurata asintoticamente.