Progetto pratico I

Ingegneria degli algoritmi

Svolto da: Luca Camilletti (0253526), Andrea Mancuso (0253152)

Obiettivo: sviluppare una struttura dati che gestisca una serie di input, suddividendoli in diverse partizioni a seconda della key e che inserisca questi elementi (key, value) nelle partizioni strutturate ad alberi AVL o LinkedList, a seconda delle condizioni indicate.

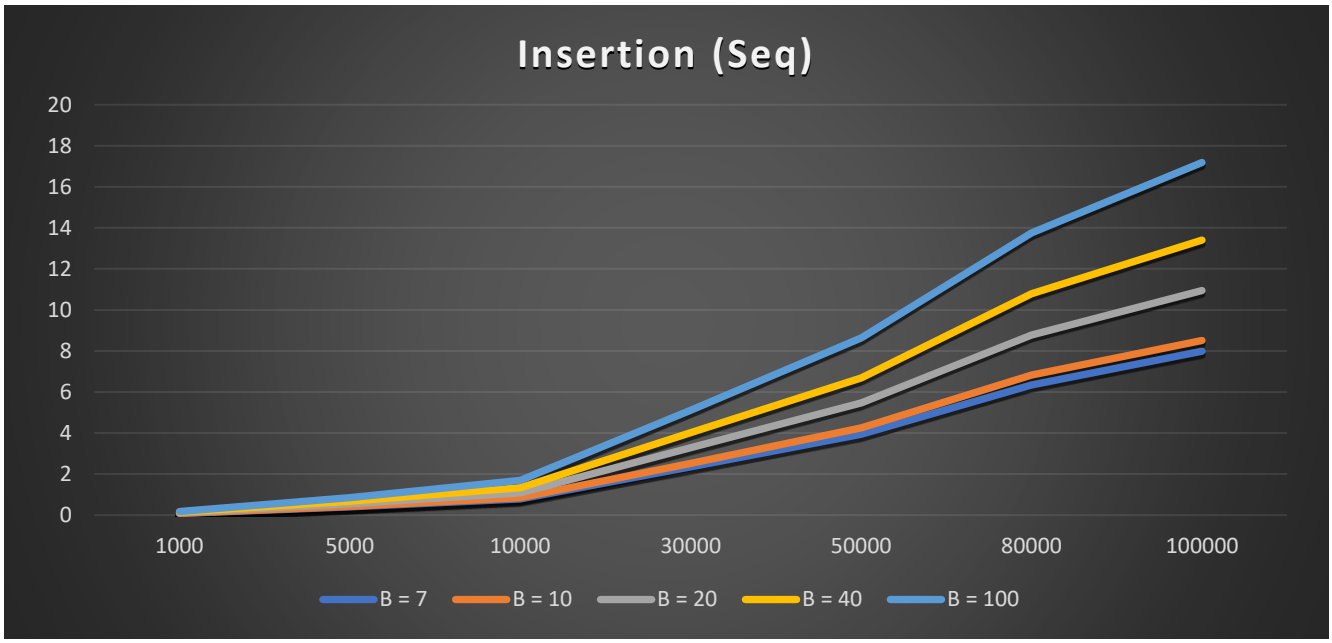
Per sviluppare il progetto abbiamo utilizzato una lista, con lo scopo di organizzare le diverse partizioni che si andranno a creare. Per ogni partizione creata abbiamo implementato, come richiesto dalla traccia, una LinkedList nel caso in cui il numero degli elementi presenti nella partizione fosse minore di sei, oppure un albero AVL, nel caso in cui il numero di elementi (presenti) nella partizione fosse maggiore di cinque. Abbiamo usato in tutto sei metodi:

- **Insert**: utilizzato per inserire gli elementi nella partizione corretta e, nel caso in cui il numero di elementi passi da 5 a 6, cambiare la lista in un albero AVL, tramite l'apposita funzione ChangeInAVL;
- **Delete**: utilizzata per cancellare un elemento presente in una delle partizioni e, nel caso in cui il numero di elementi passi da 6 a 5, cambiare l'albero AVL in una lista, tramite l'apposita funzione changeInList;
- **Search**: utilizzata per cercare un elemento all'interno della partizione, ritorna il value dell'elemento rispetto la key inserita;
- **CheckType**: utilizzata per verificare se la partizione in posizione i della lista, sia un albero AVL oppure una LinkedList. Abbiamo deciso di implementare questo metodo per rendere più snello il codice, poiché questo metodo viene richiamato sia nel metodo insert, sia nel search e nel delete;
- **ChangeInAVL** e **changeInList**: utilizzate rispettivamente per effettuare il passaggio da LinkedList ad albero AVL e da albero AVL in LinkedList. Questo cambiamento avviene all'interno dell'insert e nel delete, seguendo le condizioni stabilite inizialmente; questi due metodi sono stati implementati al fine di avere un codice pulito e comprensibile;
- **FindList**: utilizzata per trovare la partizione corretta, senza dover ripetere in ogni metodo il procedimento di controllo. Questo metodo viene implementato per rendere più snello il codice poiché, nel caso in cui non ci fosse, verrebbe ripetuto nell'insert, nel delete e nel search. In aggiunta, abbiamo deciso di modificare il modo in cui le partizioni, che si trovavano tra il massimo ed il minimo, vengono selezionate. All'inizio avevamo utilizzato un for che si interrompesse nel momento in cui avesse trovato la partizione indicata; in un secondo momento, abbiamo notato come la posizione della partizione, successivamente indicata come i, sia uguale a i = (n-min) / b, così da permetterci di non far ausilio del for e in modo tale che il tempo di esecuzione diminuisca notevolmente.

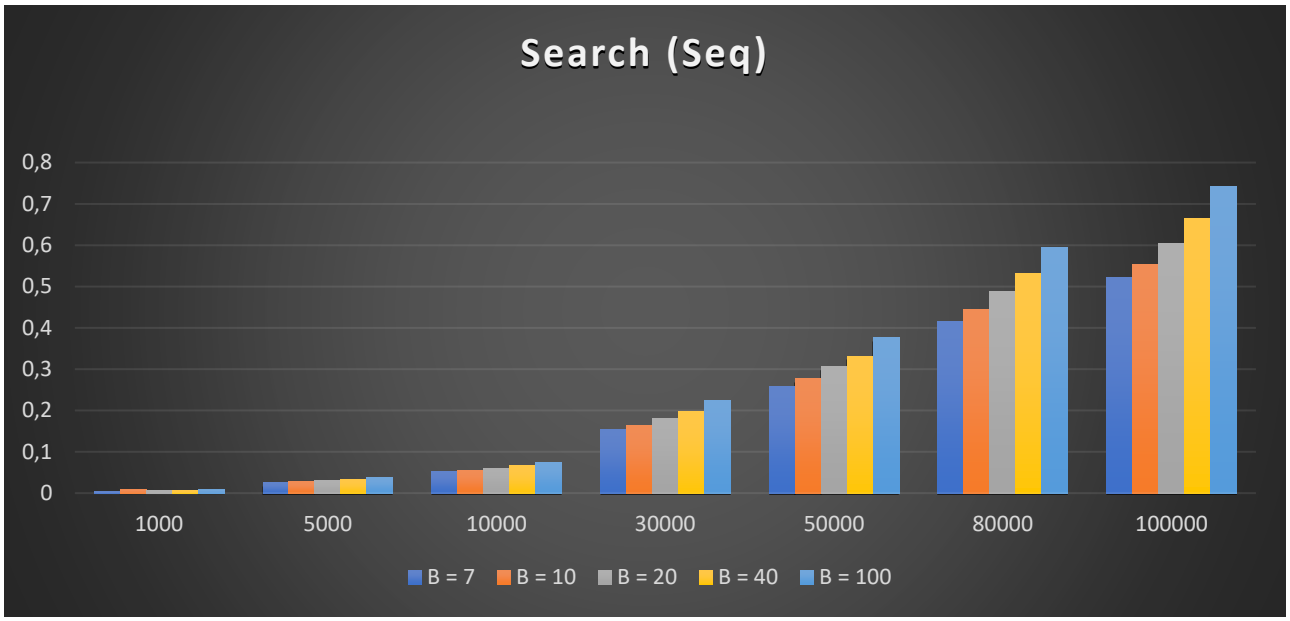
Nel progetto vengono importate due classi: LinkedListDictionary e AVLTree; abbiamo utilizzato diversi metodi, presenti nelle due classi e nelle loro superclassi, usati per gestire principalmente le LinkedList e gli alberi AVL.

Test su inserimento sequenziale

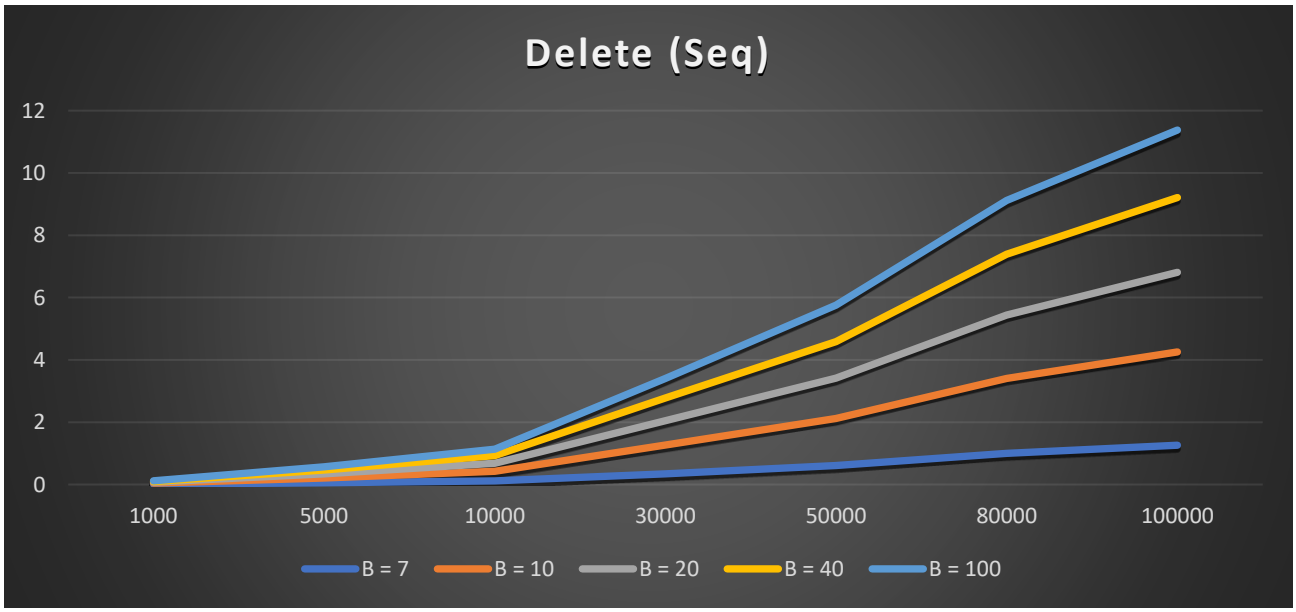
Insertion							
	Numero di elementi inseriti						
	1000	5000	10000	30000	50000	80000	100000
B = 7	0,077	0,394	0,784	2,346	3,947	6,35	7,987
B = 10	0,088	0,426	0,852	2,523	4,251	6,829	8,511
B = 20	0,117	0,555	1,091	3,283	5,464	8,783	10,944
B = 40	0,15	0,673	1,342	4,014	6,682	10,795	13,403
B = 100	0,176	0,854	1,7	5,116	8,639	13,755	17,185



Search							
	Numero di elementi inseriti						
	1000	5000	10000	30000	50000	80000	100000
B = 7	0,005	0,026	0,052	0,154	0,257	0,415	0,521
B = 10	0,008	0,027	0,055	0,163	0,277	0,444	0,553
B = 20	0,007	0,03	0,06	0,181	0,306	0,487	0,604
B = 40	0,007	0,033	0,066	0,198	0,331	0,532	0,665
B = 100	0,008	0,037	0,074	0,223	0,377	0,595	0,743

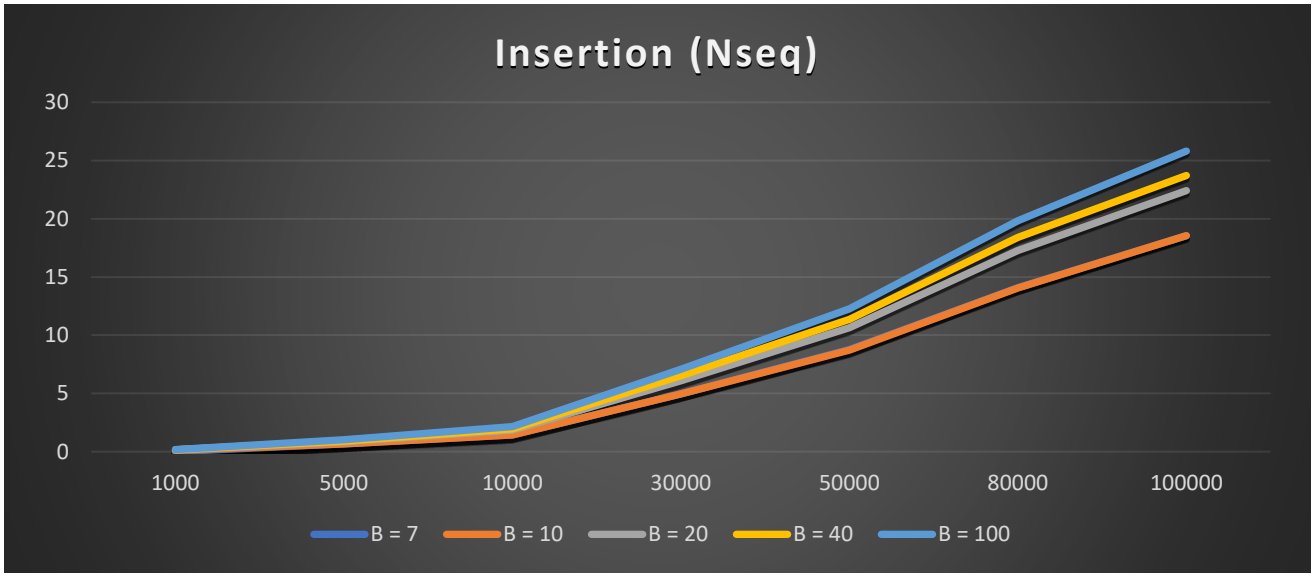


Delete							
	Numero di elementi inseriti						
	1000	5000	10000	30000	50000	80000	100000
B = 7	0,012	0,055	0,111	0,34	0,611	1,002	1,261
B = 10	0,045	0,211	0,425	1,264	2,12	3,409	4,255
B = 20	0,074	0,343	0,685	2,051	3,414	5,434	6,81
B = 40	0,094	0,466	0,926	2,776	4,588	7,388	9,21
B = 100	0,119	0,568	1,133	3,401	5,755	9,115	11,377

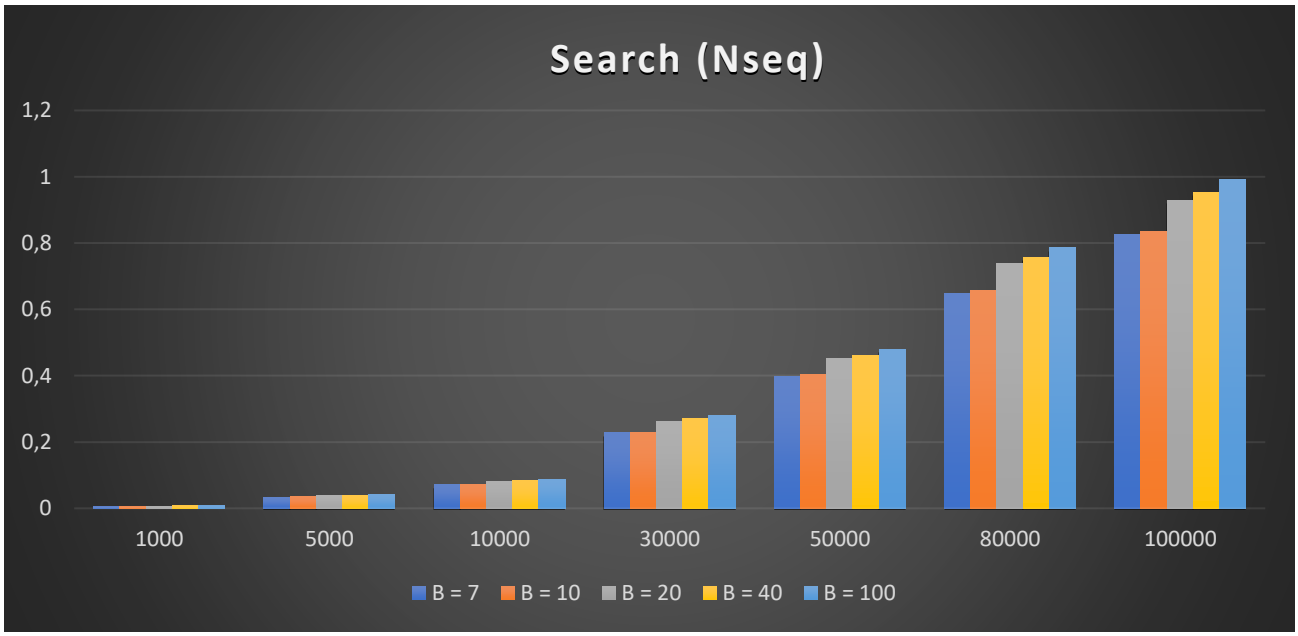


Test su un inserimento non sequenziale

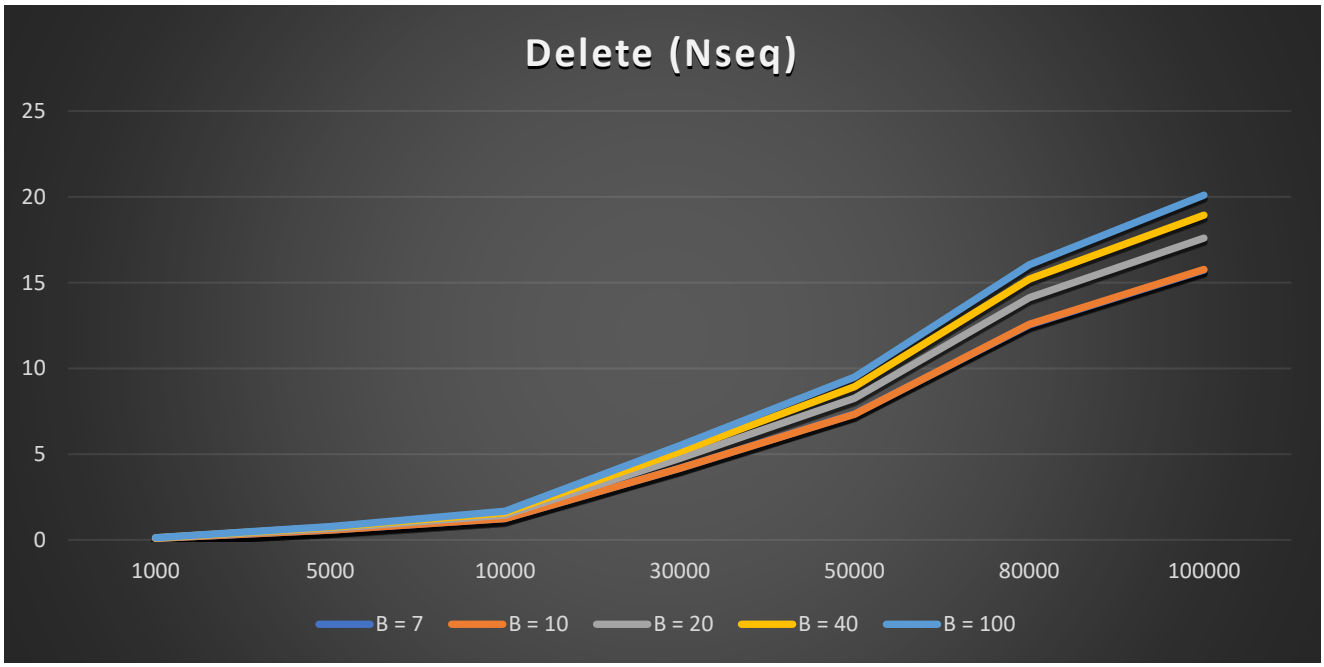
Insertion							
	Numero di elementi inseriti						
	1000	5000	10000	30000	50000	80000	100000
B = 7	0,111	0,673	1,446	4,947	8,766	14,09	18,544
B = 10	0,113	0,667	1,446	4,941	8,715	14,066	18,549
B = 20	0,151	0,859	1,841	6,095	10,65	17,271	22,415
B = 40	0,163	0,926	1,952	6,525	11,346	18,404	23,715
B = 100	0,179	1,021	2,142	7,08	12,267	19,826	25,81



Search							
	Numero di elementi inseriti						
	1000	5000	10000	30000	50000	80000	100000
B = 7	0,006	0,034	0,071	0,229	0,397	0,648	0,826
B = 10	0,006	0,035	0,072	0,23	0,404	0,657	0,834
B = 20	0,007	0,039	0,081	0,261	0,451	0,738	0,929
B = 40	0,008	0,04	0,084	0,27	0,462	0,757	0,953
B = 100	0,008	0,042	0,088	0,281	0,48	0,786	0,993



Delete							
	Numero di elementi inseriti						
	1000	5000	10000	30000	50000	80000	100000
B = 7	0,085	0,576	1,248	4,173	7,34	12,541	15,728
B = 10	0,086	0,569	1,244	4,179	7,318	12,577	15,765
B = 20	0,105	0,661	1,438	4,751	8,257	14,125	17,593
B = 40	0,118	0,73	1,572	5,142	8,932	15,195	18,931
B = 100	0,126	0,778	1,668	5,495	9,482	16,041	20,096



Commento dei risultati dei test

Per effettuare i test si è scelto di proseguire in due modi: uno con un inserimento sequenziale e uno con un inserimento non sequenziale. Per l'inserimento si è tenuta una formula precisa per la scelta di massimo e minimo. Scelta b, la quale indica la lunghezza della partizione, e avendo N (numero di elementi inseriti), abbiamo: max = int(N/b)*b + b (dove con int(N/b) si intende il casting del risultato dell'operazione interna alle parentesi) e min= b . Inserendoli in modo sequenziale, partendo da 0, otterremmo una quasi completa omogeneità nel numero di elementi per partizione. Con i test effettuati si può notare che, con questa implementazione del metodo find_list (ovvero il metodo utilizzato per trovare la partizione giusta in cui deve essere inserito l'elemento), più le partizioni sono piccole e omogenee (contenenti lo stesso numero di elementi), più sarà veloce la struttura dati ad inserire, eliminare e cercare elementi. Questo ci viene consentito dalla possibilità di trovare la partizione senza l'ausilio di un ciclo for, ma con una sola linea di codice. Ovviamente, le operazioni dentro le partizioni saranno più veloci, visto che ci sarà un numero ridotto di elementi. Quindi possiamo dedurre che la miglior situazione possibile è inserire in modo sequenziale gli elementi (per l'omogeneità del numero degli elementi per partizione), avere partizioni più piccole possibili (il cui miglior caso è b = 7) e, di conseguenza, ottenere le partizioni tutte aventi la struttura ad albero AVL, la quale è la più veloce di una linked list (nella maggior parte dei casi), da sotto tutti i punti di vista. È anche possibile implementare un algoritmo che, preso l'insieme di elementi da inserire, calcoli il miglior massimo, minimo e b (in base alle key degli elementi) al fine di ottimizzare al meglio i tempi e l'utilizzo della memoria da parte della struttura dati.