UNIVERSITÀ
DEGLI STUDI
FIRENZE

Laboratorio Algoritmi e Strutture dati

Implementazione di dizionari con varie strutture dati

Riccardo Becciolini

Maggio 2024

Indice

1 Introduzione

Lo scopo di questo esercizio è quello di confrontare varie implementazioni per un dizionario, senza prendere in considerazione la struttura dizionario predefinita in Python. Le strutture dati che sono state usate per essere messe a confronto sono: *tabella hash*, *albero binario di ricerca*, *lista concatenata*. Una volta implementate nel modo corretto per realizzare un dizionario, le strutture dati sono state testate nelle operazioni principali che possono essere utili per un dizionario. Per ciascuna struttura dati sono stati calcolati i tempi di esecuzione delle varie operazioni, confrontando poi i tempi ottenuti per le operazioni corrispondenti tra le diverse strutture.

2 Spiegazione Teorica del Problema

In questa sezione viene discusso brevemente cos'è un dizionario e le strutture dati utilizzate. Inoltre, si analizzano le complessità dei metodi delle varie strutture per giungere a ipotesi e assunzioni.

2.1 Dizionario

Un dizionario è una struttura dati astratta definito da un insieme di relazioni. Ogni relazione è composta da una coppia chiave-valore e quindi ogni chiave viene associata in maniera univoca a un valore.
Le operazioni definite nei dizionari sono:

- **Ricerca:** data una chiave restituisce il valore.
- **Inserimento:** data una chiave e un valore, se la chiave non è già presente nel dizionario, viene inserita la nuova relazione chiave-valore.
- **Eliminazione:** data una chiave rimuove la relazione chiave-valore se presente nel dizionario.

Queste saranno anche le operazioni che verranno testate.

2.2 Strutture dati usate

Di seguito sono descritte le strutture dati con le quali sono state implementate i dizionari introdotte sopra.

2.2.1 Tabelle Hash

Questa struttura dati mette in corrispondenza una data chiave a un dato valore. Quindi per sua definizione implementa un dizionario. Viene utilizzata una *funzione hash*, $h(key)$, per trasformare una chiave in un indice all'interno di un array. Quando deve essere inserita una coppia chiave-valore, viene calcolato l'indice corrispondente alla chiave tramite la funzione di hash e poi viene memorizzato il valore in quella posizione nell'array. Le operazioni di ricerca ed eliminazione si basano sempre sul trasformare la chiave in indice attraverso l'*hashing* e per poi eseguire l'operazione sul dato corrispondente.
La funzione hash e il metodo con cui vengono memorizzate le coppie chiave-valore possono avere diverse implementazioni, ciascuna delle quali gestisce le collisioni in modo diverso. Una collisione si ha quando:

$$h(k) = h(k') \;\; con \;\; k \neq k'$$

Di seguito, nella sezione dedicata all'implementazione, verranno discusse quali sono le scelte implementative adottate in questo studio.

2.2.2 Alberi Binari di Ricerca

Abbreviati con ABR, sono una struttura ad albero che è formato da un insieme di nodi. Ogni nodo è composto da dei campi tra cui la chiave k, un puntatore al figlio destro e un puntatore al figlio sinistro. Per ogni nodo x con chiave k sono rispettate le seguenti proprietà:

- Tutte le chiavi nel sottoalbero sinistro sono minori di k
- Tutte le chiavi nel sottoalbero destro sono maggiori di k

Questo comporta che l'inserimento di un nodo confronta la chiave da inserire con quella della radice. Se la chiave da inserire è minore, l'operazione viene eseguita ricorsivamente nel sottoalbero sinistro; altrimenti, nel sottoalbero destro. L'inserimento si conclude quando viene trovato un nodo nullo, momento in cui viene inserito un nuovo nodo con i campi desiderati.
L'operazione di ricerca si basa sempre sul confronto della chiave in ingresso con quelle presenti nell'albero, restituendo un risultato positivo nel caso in cui venga trovata una chiave uguale a quella in ingresso. Anche per l'eliminazione, si percorre l'albero in maniera ordinata; tuttavia, una volta trovata un'uguaglianza, viene eliminato il nodo desiderato e vengono ristabiliti i puntatori necessari.

2.2.3 Liste Concatenate

Infine questa struttura dati è formata da una sequenza di nodi, dove ognuno contiene certi campi e il puntatore al nodo successivo. L'operazione di inserimento costruisce un nodo con i campi desiderati e lo inserisce in testa alla coda. Per le altre due operazioni, viene percorsa tutta la lista fino a trovare una corrispondenza con la chiave, dopodiché viene eseguita l'operazione desiderata.

2.3 Assunti e ipotesi

Affinché gli ABR e le liste concatenate implementino un dizionario è necessario che i loro campi siano composti da chiave e valore. Inoltre deve esistere all'interno delle operazioni, dei controlli per mantenere corrispondenza univoca tra chiave e valore. Ciò potrebbe comportare un peggioramento dell'efficienza nelle operazioni delle strutture dati.

Nel contesto di questo testo, quando menzioniamo la 'complessità', ci riferiamo esclusivamente alla complessità temporale delle operazioni.

In una *tabella hash* la complessità delle operazioni dipende da quante collisioni avvengono, quindi nel caso migliore è $O(1)$. Il caso medio dipende dalla funzione hash utilizzata, nel caso in questione viene utilizzato il *metodo delle moltiplicazioni* dove anche il caso medio ha complessità $O(1)$. Il caso peggiore avviene quando ci sono il massimo numero di collisioni, quindi tutte le chiavi hanno lo stesso risultato nella funzione hash, la complessità risulta $O(n)$. Inoltre il numero di collisioni può aumentare al diminuire della dimensione dell'hash map.

Negli ABR la complessità delle operazioni corrisponde a $O(h)$, dove h è l'altezza dell'albero. Dipende tutto dall'ordine in cui sono stati inseriti le chiavi dentro l'albero. Il caso migliore è quando l'albero è bilanciato e quindi la complessità è $O(log\, n)$ che rimane uguale nel caso medio, ovvero quando i nodi sono stati inseriti in maniera casuale all'interno dell'albero. Il caso peggiore è quando le chiavi sono state inserite in ordine, decrescente o crescente, formando una catena lineare e la complessità diventa $O(n)$. In questo caso il controlli delle unicità delle chiavi vengono durante lo scorrimento dell'albero, perciò non comportano un peggioramento delle complessità.

Nelle *liste concatenate* la complessità è diversa a seconda dell'operazione, infatti per quanto riguarda l'inserimento la complessità sarebbe $O(1)$. In questo caso specifico per controllare che la chiave da inserire non sia già presente nella lista dobbiamo scorrere tutta la lista, quindi la complessità per

l'inserimento diventa $O(n)$. Per quanto riguarda la rimozione e la ricerca anche in questo caso è necessario scorrere la lista finché non viene trovata una corrispondenza, quindi la complessità è $O(n)$.

L'obiettivo di questo studio è implementare e testare le strutture dati scelte per verificare se le ipotesi corrispondono ai risultati ottenuti. Di seguito è riportata una tabella che riassume quanto detto in questa sezione. Si può notare che le complessità sono uguali per tutte le operazioni delle varie strutture dati, pertanto è stata creata un'unica tabella che riporta tutte le complessità, invece di tre tabelle separate per ciascuna operazione..

	C. caso migliore	**C. caso medio**	**C. caso peggiore**
Hash	$O(1)$	$O(1)$	$O(n)$
ABR	$O(log\, n)$	$O(log\, n)$	$O(n)$
Lista concat.	$O(n)$	$O(n)$	$O(n)$

Tabella 1: *Complessità delle operazioni di inserimento, cancellazione e ricerca*

3 Documentazione del codice

3.1 Schema del contenuto e interazioni tra moduli

Di seguito è riportato lo schema uml del codice che è stato scritto per l'implementazione delle strutture dati.

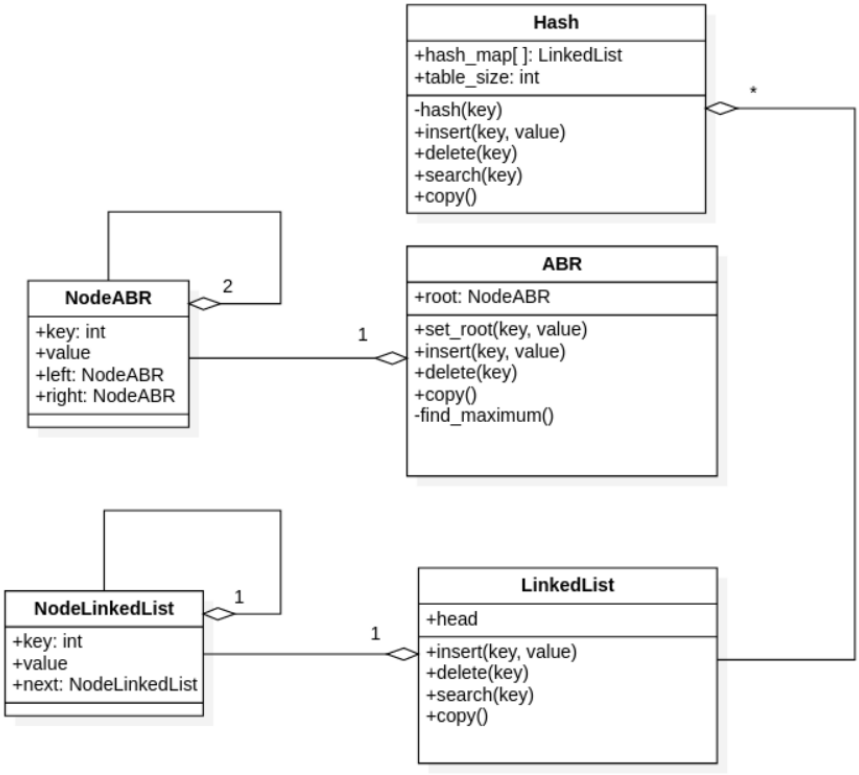

Figura 1: *Diagramma UML*

Per svolgere l'esercizio sono state implementate cinque classi: *Hash*, *ABR* e *LinkedList* sono le classi riferite alle strutture dati coinvolte. In queste classe sono compresi i metodi che implementano le operazioni delle strutture dati. Infatti nelle tre classi sono presenti i metodi *insert(key,value)*, *delete(key)* e *search(key)*. Inoltre è stato aggiunto il metodo *copy()* in ognuna delle tre classe che servirà per fare gli esperimenti di test come verrà mostrato più avanti.

Ci sono inoltre altre due classi: *NodeABR* e *NodeLinkedList* che rappresentano i nodi rispettivamente per gli alberi e le liste. Per l'hash non è necessario avere un nodo in quanto la attributo *hash_map* è composto da un vettore di dimensione fissa. Questo vettore è un vettore di liste concatenate, per questo, come riportato nella Figura 1, c'è una composizione tra la classe *Hash* e la classe *LinkedList*.

Nella classe *ABR* è stato inserito il metodo privato *find_maximum(currentNode)* questo metodo è

necessario per l'implementazione del metodo *delete*.

3.2 Scelte Implementative

La funzione hash è stata implementata attraverso il metodo *hash(key)* che è stato implementato secondo il **metodo delle moltiplicazioni**, consiste nel scegliere una costante A :

$$h(k) = \lfloor m(kA \ mod \ 1) \rfloor \ \ con \ \ A: \ 0 < A < 1$$

Dove $h(k)$ rappresenta la funzione hash per una specifica chiave k e $m = 2^p$ con p intero. Come consigliato da Knuth, è stato scelto $A \approx \frac{\sqrt{5}-1}{2}$. In questo modo la struttura dati dovrebbe avere un numero abbastanza ridotto di collisioni.
Per la gestione delle collisioni è stato scelto il **metodo concatenamento**: infatti ogni elemento dell'array hash map è una lista concatenata.

Per semplicità e per chiarezza del codice gli attributi delle classi sono tutti pubblici e non sono stati implementati i metodi *get* e *set*. È quindi possibile accedere e modificare direttamente gli attributi.

3.3 Descrizione dei metodi implementati

- **Hash**
  - **_hash(key)**: restituisce l'indice dell'array dell'hash map seguendo la logica implementativa spiegata sopra.
  - **insert(key, value)**: utilizza la funzione di hashing per trovare l'indice i nell'hash map. A quel punto controlla che la chiave non sia presente nella lista a posizione i, utilizzando il metodo *LinkedList.search(key)*. Poi inserisce in testa alla lista un nodo con chiave *key* e valore *value*. Se la chiave è già presente il nodo non viene inserito e viene stampato un messaggio.
  - **delete(key)**: utilizza la funzione di hashing per trovare l'indice i riferito alla chiave *key*. Viene cercato nodo corrispondente alla chiave *key* nella lista a posizione i dell'hash map, utilizzando il metodo *LinkedList.search(key)*. Se il nodo esiste viene rimosso altrimenti viene stampato un messaggio.
  - **search(key)**: viene utilizzata la funzione di hashing per trovare l'indice i riferita alla chiave *key*. Viene restituito il risultato del metodo *LinkedList.search(key)* della lista in posizione i nell'hash map.
  - **copy()**: viene costruito un nuovo oggetto *hash_copy*. Ogni elemento dell'hash viene copiato e inserito in *hash_copy*. Restituisce *hash_copy*.
- **ABR**:
  - **set_root(key, value)**: imposta la radice con la chiave e valore in ingresso.
  - **insert(key, value)**: se la radice è nulla chiama il metodo *set_root(key,value)* altrimenti chiama la funzione annidata *_insert_node(currentNode, key, value)* inserendo *root* come nodo corrente. Questa funzione attraversa l'albero in maniera ricorsiva fino a trovare un nodo libero e in quel caso inserisce un nuovo nodo.
  - **delete(key)**: imposta la radice chiamando la funzione annidata *_delete_node(currentNode, key)*, impostando come nodo corrente la radice stessa. Questa funzione controlla se il nodo corrente è nullo. Se la chiave da eliminare è minore della chiave del nodo corrente, la funzione cerca nel sottoalbero sinistro; se è maggiore, cerca nel sottoalbero destro. Se trova la chiave, procede con la rimozione. Se il nodo ha uno o nessun figlio, la funzione ritorna il figlio. Se ha due figli, sostituisce il nodo con il massimo valore (ultilizzando *_find_maximum()*) nel sottoalbero sinistro e lo elimina.

- **search(key)**: anche questo metodo chiama una funzione annidata *_search_node(currentNode, key)*. Questa funzione in maniera ricorsiva attraversa l'albero fino a trovare il nodo con la chiave corrispondente a *key* che restituisce come risultato.
- **copy()**: metodo che chiama la funzione annidata *_copy_node(currentNode)* per costruire l'albero di copia che viene restituito come risultato.
- **_find_maximum(currentNode)**: restituisce il nodo con chiave maggiore, attraversando l'albero a destra

- **LinkedList**
  - **insert(key, value)**: controlla che la chiave in ingresso non sia già presente nella lista usando il metodo *search(key)*. Nel caso fosse già presente viene stampato un messaggio, altrimenti viene costruito un nuovo nodo e viene messo in testa alla coda.
  - **search(key)**: viene attraversato ogni nodo della lista fino a trovare l'elemento con chiave uguale a *key*. Se il nodo viene trovato viene restituito in uscita.
  - **delete(key)**: anche in questo caso viene attraversata tutta lista finchè non viene trovato il nodo con la chiave uguale a quella in ingresso. Se trova un nodo con la chiave corrispondente, lo rimuove aggiornando i puntatori. Se il nodo è la testa, aggiorna la testa della lista. Se non trova la chiave, segnala che la chiave non è presente.
  - **copy()**: restituisce una copia della lista. Attraversa la lista e inserisce i nodi trovati nella copia.

4 Esperimenti effettuati e analisi dei risultati

4.1 Specifiche della piattaforma di test

Le specifiche della macchina usata per fare i test sono:

- **CPU**: Intel Core i7-8565U 8-Core 4.6 GHz
- **RAM**: 8GB LPDDR3
- **SSD**: 256GB M.2 NVMe PCIe 3.0
- **Sistema Operativo**: Ubuntu 24.04 LTS

La piattaforma in cui il codice è stato scritto e testa è PyCharm 2024.1.1 (Professional Edition).

4.2 Esperimenti condotti

I dati utilizzati per gli esperimenti sono hash, liste concatenate e ABR con dimensioni crescenti, riempiti con coppie chiave-valore. I valori sono casuali, mentre gli indici sono da 0 a $length - 1$, ma sono inseriti in ordine casuale in modo che che gli alberi non siano sbilanciati. Viene poi calcolato il tempo di esecuzione ogni 500 elementi inseriti, fino ad arrivare a 10000 elementi. Ogni esperimento viene ripetuto 5 volte, dopodiché viene calcolata la media dei risultati. I dati vengono poi mostrati in tabelle e grafici generati dai test utilizzando le librerie di *pandas* e *matplotlib*.
Per quanto riguarda le tabelle hash è stato scelto di costruire degli hash map di 16384 bucket, dato che deve essere della potenza del 2, richiesto dal *metodo delle moltiplicazioni*. Questa scelta è dovrebbe portare a poche collisioni così da avvicinarsi al caso medio descritto nelle ipotesi di questo esercizio.

4.3 Misurazioni

Per calcolare i tempi di esecuzione delle operazioni è stata usata la libreria **timeit** di python. Viene usata una funzione lambda per creare il contesto del timeit. La funzione timeit restituisce il tempo medio e con l'argomento *number* esprime quante volte viene eseguita la funzione per poi calcolare il tempo medio. Di seguito è mostrato come esempio l'implementazione della misurazione per l'hash.

```
# Calcolo tempi hash
hash_copy = hash.copy()
insert_hash_time = timeit.timeit(
    lambda: hash_copy.insert(random.randint(size + 1, size + 10000), random.randint( a: 0, size)), number=5)
search_hash_time = timeit.timeit(lambda: hash.search(random.randint( a: 0, size)), number=5)
delete_hash_time = timeit.timeit(lambda: hash.delete(random.randint( a: 0, size)), number=5)
insert_hash_times.append(insert_hash_time)
search_hash_times.append(search_hash_time)
delete_hash_times.append(delete_hash_time)
```

Figura 2: Esempio misurazioni

Come si può notare viene utilizzata il metodo *copy()* nella funzione lambda riguardante l'inserimento, in questo modo non viene lasciata integra la struttura dati originaria per le operazioni successive.

4.4 Risultati degli esperimenti

In questa sezione vengono mostrate le tabelle e i grafici risultanti dagli esperimenti. Per ciascuna operazione vengono mostrati la tabella e il grafico dei tempi per ogni struttura dati. Verranno discussi e confrontati nella sezione *Conclusioni*.

4.4.1 Inserimento

Tabella inserimento hash

# Elementi	Tempo(s)
500	1.1781e-05
1000	1.2432e-05
1500	1.4805e-05
2000	1.3476e-05
2500	1.5485e-05
3000	1.4037e-05
3500	1.3372e-05
4000	1.4004e-05
4500	1.4503e-05
5000	5.7358e-05
5500	1.6916e-05
6000	1.4342e-05
6500	1.4503e-05
7000	1.6063e-05
7500	1.5427e-05
8000	1.5554e-05
8500	1.7097e-05
9000	1.8390e-05
9500	1.6341e-05
10000	1.7922e-05

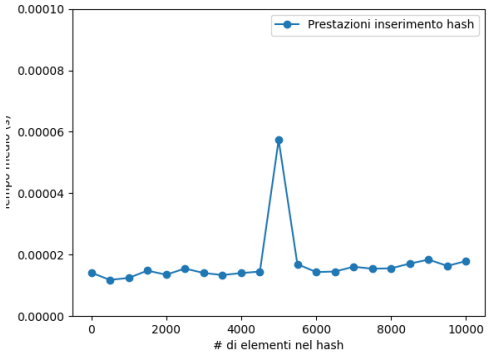

Figura 3: Tempi calcolati per la tabella hash

Tabella inserimento albero binario di ricerca

# Elementi	Tempo(s)
500	1.3140e-05
1000	1.2197e-05
1500	1.1406e-05
2000	1.1997e-05
2500	1.4631e-05
3000	1.3619e-05
3500	2.6333e-05
4000	1.2782e-05
4500	1.5988e-05
5000	1.4717e-05
5500	1.5075e-05
6000	1.4654e-05
6500	1.5327e-05
7000	1.8070e-05
7500	1.7758e-05
8000	1.5655e-05
8500	1.5561e-05
9000	1.6817e-05
9500	1.8118e-05
10000	1.9947e-05

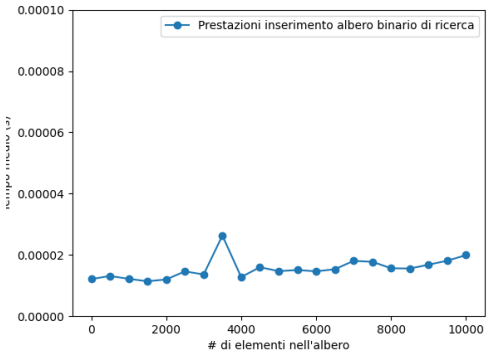

Figura 4: Tempi calcolati per l'albero binario di ricerca

Tabella inserimento lista cancatenata

# Elementi	Tempo(s)
500	1.7916e-04
1000	4.2618e-04
1500	4.4625e-04
2000	7.7592e-04
2500	6.8156e-04
3000	7.2462e-04
3500	6.8971e-04
4000	7.5433e-04
4500	8.6675e-04
5000	9.4333e-04
5500	1.0566e-03
6000	2.2862e-03
6500	1.2859e-03
7000	1.3150e-03
7500	1.4229e-03
8000	1.5025e-03
8500	1.5987e-03
9000	3.1317e-03
9500	1.7985e-03
10000	1.9256e-03

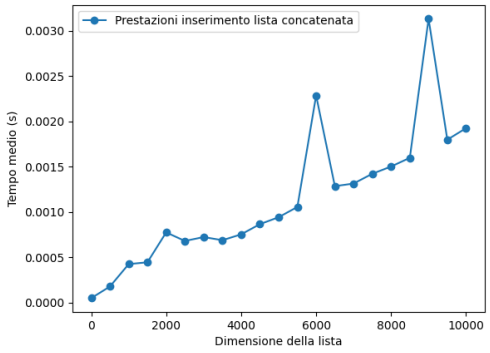

Figura 5: Tempi calcolati per la lista concatenata

4.4.2 Ricerca

Tabella ricerca hash

# Elementi	Tempo(s)
500	5.8590e-06
1000	6.3300e-06
1500	6.5390e-06
2000	6.5120e-06
2500	6.4090e-06
3000	6.8090e-06
3500	6.4240e-06
4000	5.9730e-06
4500	6.7690e-06
5000	6.5140e-06
5500	6.3510e-06
6000	7.5750e-06
6500	6.8670e-06
7000	1.3517e-05
7500	6.6110e-06
8000	6.6110e-06
8500	7.5290e-06
9000	7.8890e-06
9500	7.1290e-06
10000	6.7180e-06

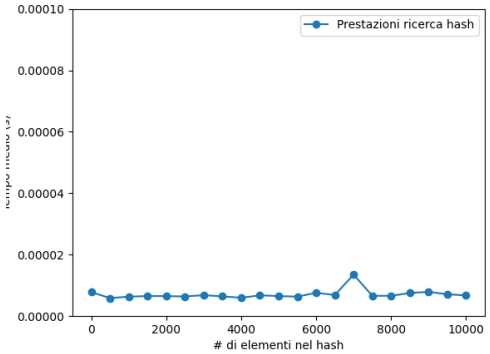

Figura 6: Tempi calcolati per la tabella hash

Tabella ricerca albero binario di ricerca

# Elementi	Tempo(s)
500	8.6560e-06
1000	8.9020e-06
1500	8.9580e-06
2000	9.4660e-06
2500	9.9250e-06
3000	1.0481e-05
3500	1.0087e-05
4000	9.9520e-06
4500	1.2308e-05
5000	1.0824e-05
5500	1.2025e-05
6000	1.1583e-05
6500	1.5287e-05
7000	1.2386e-05
7500	1.2631e-05
8000	1.4085e-05
8500	1.3525e-05
9000	1.3524e-05
9500	1.2673e-05
10000	1.6087e-05

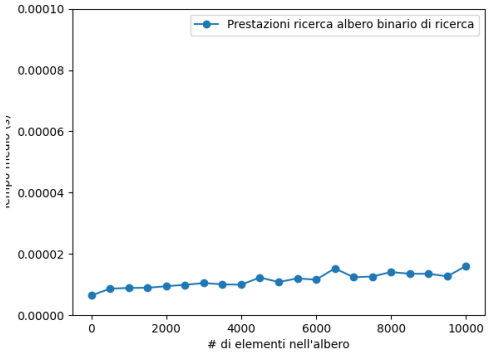

Figura 7: Tempi calcolati per l'albero binario di ricerca

Tabella ricerca lista cancatenata

# Elementi	Tempo(s)
500	6.5897e-05
1000	6.0042e-05
1500	9.0368e-05
2000	3.6740e-04
2500	2.1684e-04
3000	1.7128e-04
3500	3.7431e-04
4000	6.0185e-04
4500	5.2598e-04
5000	7.7915e-04
5500	7.0363e-04
6000	7.6127e-04
6500	1.3131e-03
7000	1.1519e-03
7500	1.1483e-03
8000	1.3409e-03
8500	1.3630e-03
9000	1.6867e-03
9500	1.4610e-03
10000	2.1568e-03

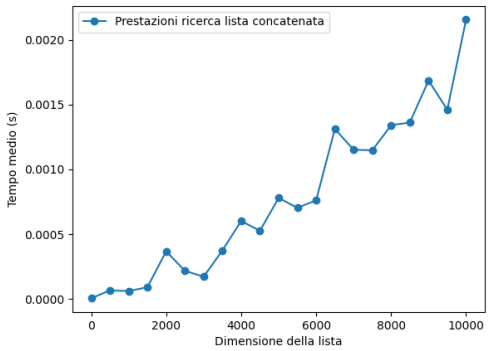

Figura 8: Tempi calcolati per la lista concatenata

4.4.3 Cancellazione

Tabella eliminazione hash

# Elementi	Tempo(s)
500	7.1320e-06
1000	7.4260e-06
1500	7.5610e-06
2000	7.9290e-06
2500	8.8980e-06
3000	8.1210e-06
3500	7.4940e-06
4000	7.7610e-06
4500	8.0590e-06
5000	8.2760e-06
5500	7.8730e-06
6000	7.7510e-06
6500	8.4470e-06
7000	8.3450e-06
7500	8.2330e-06
8000	8.7920e-06
8500	9.0070e-06
9000	9.3450e-06
9500	9.5330e-06
10000	8.9810e-06

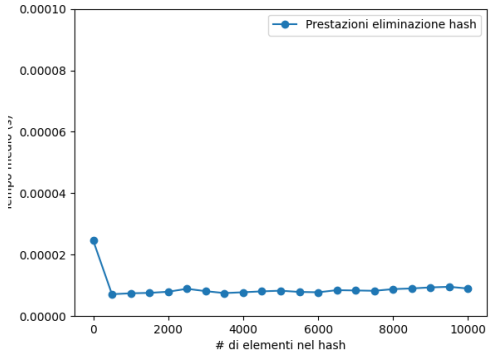

Figura 9: Tempi calcolati per la tabella hash

Tabella eliminazione albero binario di ricerca

# Elementi	Tempo(s)
500	1.2495e-05
1000	1.1393e-05
1500	1.3368e-05
2000	1.1962e-05
2500	1.2593e-05
3000	1.3363e-05
3500	1.3353e-05
4000	1.2353e-05
4500	1.5068e-05
5000	2.7135e-05
5500	1.5106e-05
6000	1.5479e-05
6500	1.5034e-05
7000	1.7375e-05
7500	1.5487e-05
8000	1.6172e-05
8500	1.6323e-05
9000	1.8494e-05
9500	1.7992e-05
10000	1.8818e-05

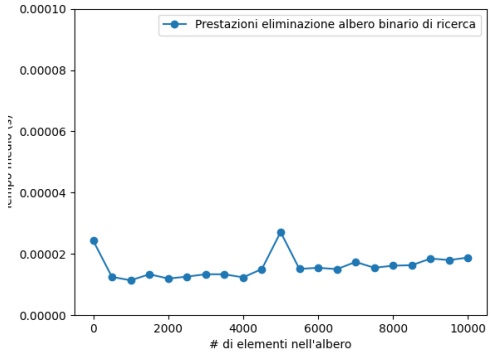

Figura 10: Tempi calcolati per l'albero binario di ricerca

Tabella eliminazione lista cancatenata

# Elementi	Tempo(s)
500	3.1113e-05
1000	7.7337e-05
1500	1.3148e-04
2000	8.3476e-05
2500	2.9227e-04
3000	2.4985e-04
3500	3.3186e-04
4000	3.2705e-04
4500	2.0370e-04
5000	3.9956e-04
5500	5.0518e-04
6000	4.2565e-04
6500	4.9382e-04
7000	4.4824e-04
7500	6.4846e-04
8000	7.8821e-04
8500	8.3954e-04
9000	7.4733e-04
9500	9.1307e-04
10000	1.2331e-03

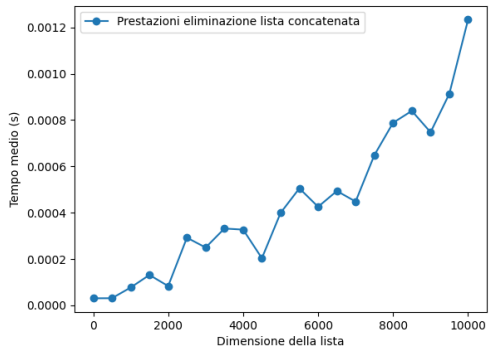

Figura 11: Tempi calcolati per la lista concatenata

5 Conclusioni

I risultati degli esperimenti confermano le ipotesi iniziali, in particolare:

- **Operazioni tabelle hash**: (Figure 3, 6 e 9) dalle tabelle possiamo notare che l'andamento temporale all'aumentare degli elementi nell'hash rimane costante in tutte le operazioni. Questo è anche confermato dai grafici.
- **Operazioni alberi binari di ricerca**: (Figure 4,7 e 10) in questo caso come si può vedere dalle tabelle e dai grafici sembra che l'andamento temporale ha una leggera crescita all'aumentare degli elementi all'interno dell'ABR. Possiamo dedurre che si tratti di un andamento logaritmico, dato che la crescita rimane molto contenuta.
- **Operazioni lista concatenata**: (Figure 5, 8 e 11) da questi risultati notiamo che la crescita temporale all'aumentare degli elementi è molto più vertiginosa rispetto agl'altri casi. Analizzando la tabella e i grafici si può dedurre che tengono un andamento lineare.

I risultati ottenuti sono coerenti con le complessità temporali ipotizzate inizialmente. In conclusione, le strutture dati *hash* e *ABR* risultano più adatte per implementare i dizionari rispetto alla *lista concatenata*, che si dimostra meno efficiente anche in presenza di pochi elementi.
Confrontando *hash* e *ABR*, si può osservare che la prima è più efficiente, poiché presenta un andamento costante. Tuttavia, in questo studio non è stata considerata la complessità spaziale. Per minimizzare le collisioni nell'hash, è stato utilizzato un array molto grande, che probabilmente occupa molta memoria. Di conseguenza, ci sono situazioni in cui l'ABR potrebbe risultare più vantaggioso in condizioni di memoria limitata, anche se questo aspetto non è stato esaminato in dettaglio nel presente studio.