UNIVERSITÀ
DEGLI STUDI
FIRENZE

Università degli Studi di Firenze

Dipartimento di Ingegneria dell'Informazione

Laboratorio di Algoritmi e Strutture Dati

Autore:
Niccolò Scommegna

N° Matricola:
7054502

Corso principale:
Algoritmi e Strutture Dati

Docente corso:
Simone Marinai

Novembre-Dicembre 2023

Indice

Elenco delle figure

Elenco delle tabelle

1 Introduzione

1.1 Obiettivi dell'esercitazione

Nella seguente relazione verranno analizzate le differenze tra diverse implementazioni di code di priorità. Nello specifico le tre tipologie di strutture dati prese in considerazione sono:

- Heap binario
- Lista concatenata
- Lista concatenata ordinata

Gli obbiettivi dell'esercitazione sono quelli di ricreare le strutture dati appena indicate, successivamente eseguire dei test appropriati per determinare i vantaggi e gli svantaggi di ogni tipologia di coda, e infine confrontare i risultati ottenuti dai test con i risultati teorici che sono stati osservati durante il corso di Algoritmi e Strutture Dati.

1.2 Struttura della relazione

Durante lo sviluppo della relazione sono state affrontate le seguenti tematiche per esporre al meglio lo svolgimento dell'esercitazione:

- **Descrizione teorica del contesto:** breve illustrazione delle strutture dati e dei risultati attesi delle operazioni basandosi sulle nozioni teoriche del libro di testo del corso.[1]
- **Documentazione del codice e scelte implementative:** verrà affrontato l'organizzazione delle classi e discusso rapidamente i vari metodi che le compongono.
- **Descrizione dei test eseguiti:** saranno analizzati gli esperimenti svolti e i risultati che ne derivano.
- **Riflessioni finali:** riflessione sui dati ottenuti dai test con l'intento di confrontarli con i risultati teorici.

1.3 Specifiche della piattaforma di test

In questa sezione sono elencate le caratteristiche del computer su cui è stata svolta l'esercitazione:

- **CPU:** 3 GHz Intel Core i5 6 core
- **RAM:** 8 GB 2667 MHz DDR4
- **SSD:** APPLE SSD SM0032L 1,03TB

Il linguaggio di programmazione utilizzato è **Phyton** ed è stato usato l'IDE **PyCharm Professional 2023.2.5**. Per la scrittura della relazione è stato utilizzato l'editor online **Overleaf**.

2 Descrizione teorica del contesto

2.1 Le code di priorità

Una coda di priorità è una struttura dati che serve a mantenere un insieme dinamico di elementi, ciascuno con un valore associato detto chiave. Si differenzia da una coda e da uno stack perché ogni elemento contenuto nell'insieme possiede un valore di priorità e in base a questa si può definire un ordinamento specifico e apposite operazioni da svolgere. Al contrario, una coda e uno stack si basano su particolari principi: rispettivamente *First-In, First-Out* e *Last-In, First-Out*, quindi le operazioni di inserimento e cancellazione sono predeterminate.

Esistono due tipi di code di priorità: quelle di max-priorità e quelle di min-priorità. Nelle code di max-priorità l'elemento più importante, quello a priorità maggiore, è quello associato alla chiave con valore più grande, al contrario nella coda di min-priorità l'elemento più importante è quello associato alla chiave con valore minore. Le operazioni che offrono le due code di priorità hanno gli stessi scopi, ma si adattano alle proprie caratteristiche. La varietà che offrono le code di priorità consente di poterle adattare ad un gran numero di applicazioni e situazioni. In termini di efficienza non esiste un tipo di coda (tra max e min priorità) che sia più vantaggioso dell'altro, quindi per lo svolgimento dell'elaborato è stata scelta in modo arbitrario la coda di max-priorità. Quest'ultima supporta le seguenti operazioni:

- Insert(S, x) inserisce l'elemento x nell'insieme S, che equivale all'operazione: $S = S \cup \{x\}$.
- Maximum(S) restituisce l'elemento di S con la chiave più grande.
- Extract-Max(S) rimuove e restituisce l'elemento di S con la chiave più grande.
- Increase-Key(S, x, k) aumenta il valore della chiave dell'elemento x al nuovo valore k, che si suppone sia almeno grande quanto il valore corrente della chiave dell'elemento x.

Le operazioni che sono state scelte per eseguire i test sono quella di inserimento e di estrazione, perché sono le azioni che vengono usate più frequentemente quando si ha a che fare con le code di priorità.

2.1.1 Heap binario

Un heap binario è una struttura dati composta da un array che possiamo considerare come un albero binario quasi completo, come mostra la Figura 1. Un albero binario si dice quasi completo se tutti i livelli sono completi, tranne al più l'ultimo che può essere riempito da sinistra fino a un certo punto. Ogni nodo dell'albero corrisponde a un elemento nell'array, per comodità chiameremo l'array A. La radice dell'albero è situata in $A[0]$ e se i è l'indice di un nodo, gli indici di suo padre, del figlio sinistro e del figlio destro possono essere calcolati facilmente nei seguenti modi:

- Parent$(i) = \lfloor (i-1)/2 \rfloor$
- Left$(i) = 2i + 1$
- Right$(i) = 2i + 2$

Analogamente alle code di priorità, esistono due tipi di heap binari: **max-heap** e **min-heap**. In entrambi i casi i valori nei nodi soddisfano una proprietà[1] dell'heap:

- $A[\text{Parent}(i)] \geq A[i]$ per i **max-heap** (come quello in Figura 1). Ciò comporta che il valore di un nodo è al massimo il valore di suo padre, perciò l'elemento con il valore più grande è memorizzato nella radice.
- $A[\text{Parent}(i)] \leq A[i]$ per i **min-heap**. In questo caso si ha una logica opposta al precedente e quindi nella radice è memorizzato l'elemento con il valore più piccolo.

[1] Proprietà valide per ogni nodo i diverso dalla radice.

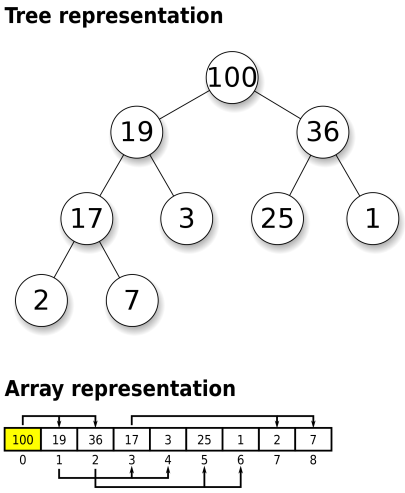

Figura 1: Raffigurazione di un max-heap sia come albero binario che come array

Infine definiamo cos'è l'altezza di un nodo e di un heap e perché queste nozioni sono rilevanti nell'analisi delle sue prestazioni. L'altezza di un nodo è il numero di archi nel cammino semplice più lungo che dal nodo scende fino a una foglia, mentre l'altezza di un heap è l'altezza della sua radice. Poiché un heap di n elementi è basato su un albero binario completo, la sua altezza è $\Theta(\lg n)$. Come vedremo più approfonditamente nei paragrafi 2.2.1 e 2.2.2, le operazioni di inserimento ed estrazione in un heap vengono eseguite in un tempo che è al massimo proporzionale all'altezza dell'albero, quindi richiedono un tempo $O(\lg n)$.

2.1.2 Lista concatenata

Una lista concatenata è una struttura dati i cui elementi sono disposti in ordine lineare come in un array, ma a differenza di quest'ultimo, dove l'ordine è mantenuto dagli indici dell'array stesso, in una lista concatenata ogni elemento contiene anche l'informazione su chi sia l'elemento successivo. Come mostra la Figura 2, ogni elemento prende il nome di *nodo* ed è composto da due campi: il campo *data*, che contiene il valore dell'elemento, e il campo *next*, che contiene l'informazione sul prossimo elemento. Inoltre notiamo che il primo oggetto della lista prende il nome di *head* e punta al primo nodo della lista, mentre il campo *next* dell'ultimo nodo punta al valore *Null* che ha lo scopo di indicare la fine della lista.

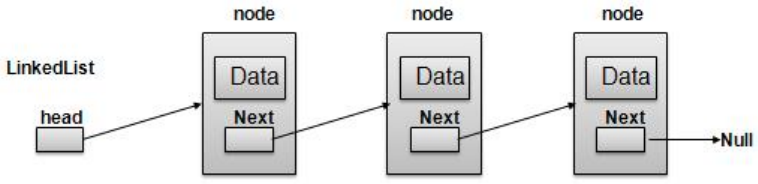

Figura 2: Raffigurazione di una lista concatenata

2.1.3 Lista concatenata ordinata

Una lista concatenata ordinata è a tutti gli effetti una lista concatenata dove però i nodi della lista sono organizzati in ordine crescente o decrescente, in base al valore contenuto nel campo *data*. Per mantenere questa proprietà si deve però porre maggiore attenzione all'operazione di inserimento che dovrà cercare la corretta posizione del nuovo elemento.

2.2 Operazioni principali e risultati attesi

In questo paragrafo tratteremo in modo più approfondito le operazioni di inserimento ed estrazione, e come queste vengono eseguite su ognuna delle tre strutture dati descritte precedentemente. Da ora in poi nella relazione, quando si parlerà della struttura dati heap faremo riferimento, nello specifico, a un max-heap, mentre quando parleremo di lista concatenata ordinata faremo riferimento ad una lista concatenata con ordinamento decrescente, poiché queste sono le tipologie scelte per eseguire i test. Come detto precedentemente nel Paragrafo 2.1 la scelta tra le due tipologie di heap non comporta nessun vantaggio in termini di efficienza, si tratta perciò di una scelta totalmente arbitraria come quella sul tipo di ordinamento della lista concatenata ordinata.

2.2.1 Operazione di inserimento

- **Heap:** Inizialmente si inserisce il nuovo elemento in coda all'array e si tiene traccia del suo indice in una variabile temporanea, successivamente si controlla che la proprietà del max-heap sia rispettata. Nel caso in cui questa lo sia, ovvero quando il valore dell'elemento inserito è minore del valore del genitore, l'operazione termina; altrimenti si esegue un ciclo `while` per ripristinare la proprietà tramite un approccio *bottom-up*, nel quale si scambia l'elemento appena inserito con il genitore fino a quando il nuovo elemento raggiunge la posizione corretta. Il passaggio di ripristino della proprietà dell'heap viene eseguita scorrendo l'albero lungo la sua altezza, perciò la complessità temporale dell'operazione sarà logaritmica.

- **Lista concatenata:** Si crea un nuovo `Node` con il valore passato alla funzione e si aggiunge in testa alla lista aggiornando il campo `head` della struttura dati. È un'operazione che richiede un tempo costante per essere eseguita.

- **Lista concatenata ordinata:** Anche in questo caso si crea un nuovo oggetto `Node` ma prima di inserire il nuovo elemento nella lista si deve trovare la posizione corretta per mantenere la lista ordinata in modo decrescente, perciò si deve scorrere tutta la struttura dati finché non si trova un elemento che ha il campo `data` con valore minore o uguale al valore passato alla funzione, a questo punto si aggiornare il campo `next` del nodo precedente e del nuovo nodo per mantenere l'ordinamento. La ricerca della corretta posizione viene eseguita scorrendo ogni elemento della lista, perciò l'operazione di inserimento ha una complessità lineare.

Nella Tabella 1 è possibile confrontare le complessità temporali dell'operazione di inserimento per le tre code di priorità.

2.2.2 Operazione di estrazione

- **Heap:** In questa struttura dati l'elemento con priorità massima è situato nella radice perciò viene immediatamente salvato in una variabile temporanea che verrà restituita al termine dell'operazione. Prima di restituire il massimo, si copia il valore dell'ultimo elemento nella radice e si rimuove tale elemento dalla struttura dati, successivamente si procede, tramite un approccio *top-down*, a ripristinare la proprietà del max-heap. Nel processo di ripristino si chiama il metodo `max_heapify(i)`, che nel caso specifico dell'estrazione del valore massimo gli viene passato il valore `i=0`, quindi questa procedura inizia dalla radice. Come vedremo successivamente nel Paragrafo 3.3.2, la procedura `max_heapify(i)` opera sulla lunghezza dell'albero, impiega quindi un tempo logaritmico per ripristinare la proprietà. Nel complesso anche l'operazione di estrazione impiega un tempo logaritmico, in quanto svolge una quantità costante di lavoro oltre alla chiamata del metodo `max_heapify(i)`.

- **Lista concatenata:** Si deve scorrere tutta la lista per cercare l'elemento con il valore maggiore e rimuoverlo, per fare questo si tiene traccia nella variabile `previous_max_node` il nodo precedente al nodo che contiene il valore maggiore cosicché, una volta terminata la fase di ricerca si può direttamente modificare il campo `next` del nodo `previous_max_node` facendolo puntare al nodo successivo del nodo contenente il massimo. L'intera operazione richiede sempre di controllare ogni elemento della lista perciò avrà una complessità lineare.

- **Lista concatenata ordinata:** Anche in questa struttura dati l'elemento con valore maggiore è memorizzato in testa perché si mantiene un ordinamento decrescente, perciò l'estrazione di questo elemento si limita a riassegnare il campo `head` della lista all'elemento successivo e a restituire il valore contenuto precedentemente in testa alla lista. In questo caso l'operazione richiedete un tempo costante per essere eseguita.

Sempre nella Tabella 1 è possibile confrontare le complessità temporali dell'operazione di estrazione per le tre code di priorità.

	Heap	Lista concatenata	Lista concatenata ordinata
Inserimento	$O(\lg\ n)$	$O(1)$	$O(n)$
Estrazione	$O(\lg\ n)$	$\Theta(n)$	$O(1)$

Tabella 1: Complessità temporale delle operazioni di inserimento e estrazione

3 Documentazione del codice e scelte implementative

In questa sezione approfondiremo l'implementazione del codice utilizzato per la realizzazione dei test e dei grafici, scritto in linguaggio Python. Suddivideremo l'argomento in tre sottosezioni: organizzazione delle classi, scelte implementative e descrizione dei metodi.

3.1 Organizzazione delle classi

L'organizzazione delle classi è descritta dal class diagram in Figura 3. Ho scelto di implementare l'interfaccia `PriorityQueueInterface` che rappresenta la coda di priorità e definisce le operazioni principali, ovvero `insert(item)` e `extract_max()`, che saranno implementate dalle classi che la estendono. Inoltre è caratterizzata dall'attributo `size` che mantiene il numero di elementi memorizzati all'interno della coda. La classe `Heap` rappresenta l'omonima struttura dati, implementa le operazioni definite dall'interfaccia e altri metodi necessari per rappresentare la logica di questa coda di priorità. La classe `LinkedList` ha lo stesso scopo della classe precedente ma rappresenta una lista concatenata. Infine la classe `OrderedLinkedList` estende ulteriormente `LinkedList` poiché definisce un caso particolare di lista concatenata, ed è stato sufficiente modificare i metodi di inserimento ed estrazione per rappresentare questa struttura dati. Entrambe le liste concatenate definiscono un rapporto di composizione con la classe `Node` che rappresenta i nodi delle liste, composti dai campi `data` e `next`, infatti le liste mantengono un riferimento alla testa della coda tramite l'attributo `Head` che è di tipo `Node`. In conclusione anche la classe `Node` definisce una rapporto di composizione con se stessa perché il campo `next` mantiene il riferimento al nodo successivo.

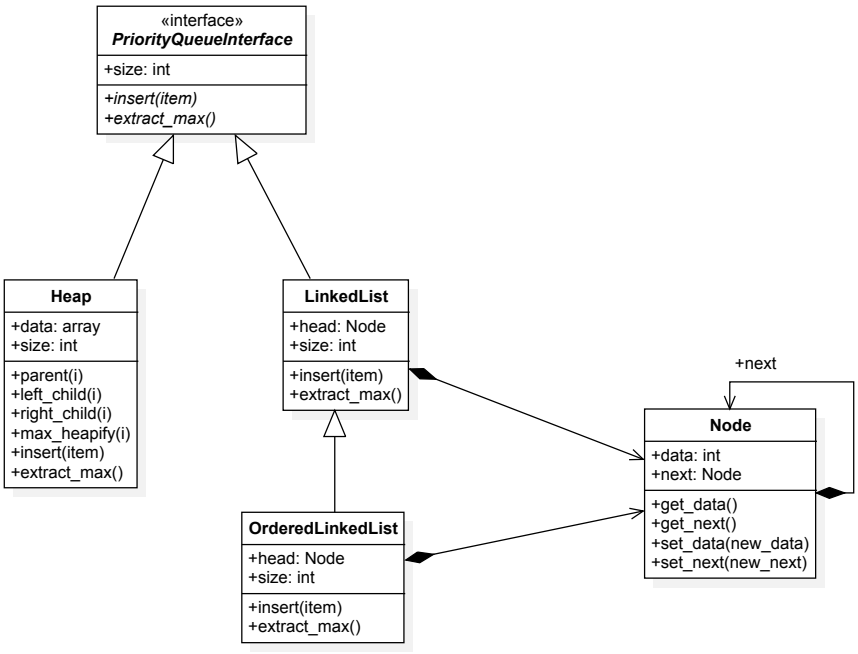

Figura 3: Class diagram

3.2 Scelte implementative

Come accennato nei paragrafi precedenti è stato scelto di implementare code di max-priorità, perciò la classe `Heap` rappresenta nello specifico un max-heap, e la classe `OrderedLinkedList` ordina gli elementi in modo decrescente, garantendo così che l'elemento con il campo `data` più grande, e quindi con priorità maggiore, sia posizionato in testa alla lista. Inoltre l'implementazione della struttura dati heap è realizzata tramite un array. In Python, l'indicizzazione degli array inizia da 0, a differenza dello pseudocodice presente nel libro di testo che invece inizia da 1, perciò i metodi implementati sono stati leggermente modificati, pur mantenendo lo stesso scopo.

3.3 Descrizione dei metodi

In questa sezione oltre a descrivere i metodi delle varie classi verranno descritti anche il file `main.py`, che ha lo scopo principale di implementare i test, e il file `enums.py`, che è una raccolta delle enumerazioni utilizzate per gestire gli input da parte dell'utente.

3.3.1 `PriorityQueueInterface`

- `insert(item)`: inserisce l'elemento `item` nella coda di priorità
- `extract_max()`: estrae l'elemento con priorità massima dalla coda di priorità e lo restituisce

3.3.2 `Heap`

- `parent(i)`: calcola l'indice del padre del nodo con indice `i`
- `left_child(i)`: calcola l'indice del figlio sinistro del nodo con indice `i`
- `right_child`: calcola l'indice del figlio destro del nodo con indice `i`
- `max_heapify(i)`: subrutine che viene chiamata dal metodo `extract_max()` per ripristinare la proprietà del max-heap ed opera nel seguente modo. Ad ogni passo viene determinato l'elemento massimo tra `A[i]`, `A[left_child(i)]` e `A[right_child(i)]`. Se `A[i]` è il massimo allora il sottoalbero con radice nel nodo `i` rispetta la proprietà. Altrimenti uno dei figli è l'elemento più grande, quindi `A[i]` viene scambiato con il figlio con valore massimo, in questo modo, il nodo `i` e i suoi figli soddisfano la proprietà. Adesso però il nodo che originariamente era in `i` è stato scambiato al livello inferiore e dobbiamo controllare che anche questo rispetti la proprietà con il proprio sottoalbero, perciò chiameremo ricorsivamente il metodo `max_heapify(i)` per questo sottoalbero. Il processo di ripristino della proprietà viene eseguita scorrendo l'albero lungo la sua lunghezza e quindi la complessità temporale sarà logaritmica.
- `insert(item)`: esegue l'operazione di inserimento descritta nel Paragrafo 2.2.1
- `extract_max()`: esegue l'operazione di estrazione descritta nel Paragrafo 2.2.2

3.3.3 `LinkedList` e `OrderedLinkedList`

Entrambe le classi espongono solo i metodi che definiscono le operazioni di inserimento e estrazione che sono già stati descritti nei Paragrafi 2.2.1 e 2.2.2.

3.3.4 `Node`

- `get_data()`: restituisce il valore di priorità del nodo
- `get_next()`: restituisce il riferimento al nodo successivo
- `set_data(new_data)`: imposta il valore di priorità del nodo a `new_data`
- `set_next(new_next)`: imposta il riferimento al nodo successivo a `new_next`

3.3.5 File `enums.py`

- `Class DataType`: enumerazione dei tre tipi di coda di priorità
- `Class ArrayType`: enumerazione dei tre tipi di array in input utilizzati per svolgere i test (random, ordinato crescente, ordinato decrescente)
- `Class ArraySize`: enumerazione delle tre grandezze degli array in input utilizzati per svolgere i test (100 elementi, 1000 elementi, 10000 elementi)

3.3.6 File `main.py`

In questo file sono presenti varie funzioni ausiliarie realizzate per ottenere gli input dall'utente, per svolgere i test e generare i grafici.

- `get_data_structure_type()`: permette all'utente di scegliere la struttura dati da testare
- `get_array_type()`: permette all'utente di scegliere il tipo di array in input da usare durante il test
- `get_array_size()`: permette all'utente di scegliere la grandezza dell'array in input da usare durante il test
- `create_data_structure(data_structure_type)`: crea la struttura dati in base alla scelta dell'utente
- `create_numpy_array(array_type, array_size)`: crea l'array in input in base alle scelte dell'utente
- `perform_insertion(...)`: effettua gli inserimenti nella struttura dati, calcola i tempi di ciascun inserimento e li registra in una matrice, che rappresenta l'oggetto restituito dalla funzione
- `perform_ectraction(...)`: effettua le estrazioni dalla struttura dati, calcola i tempi di ciascuna estrazione e li registra in una matrice, che rappresenta l'oggetto restituito dalla funzione
- `plot_operation_times(...)`: funzione utilizzata per generare grafici e presenta quattro varianti, ciascuna adatta a una tipologia specifica di test. Sebbene varino alcune informazioni, come i valori massimi della scala e i dettagli relativi al test, tutte operano in modo ananlogo

Le funzioni precedentemente elencate sono usate all'interno del metodo `main()` che, a seconda degli input dell'utente, gestisce il flusso degli inserimenti e delle estrazioni dalle strutture dati e genera i relativi grafici. L'elenco successivo di funzioni ausiliarie si differenzia dal precedente perché sono utilizzate dal metodo `mean_and_medain()` che ha lo scopo di calcolare la media e la mediana dei tempi per effettuare un ciclo completo di inserimenti e di estrazioni dalle strutture dati.

- `perform_insertion_mean_median(...)`: effettua gli inserimenti nella struttura dati, calcola il tempo per effettuare tutti gli inserimenti e ne restituisce il valore
- `perform_extraction_mean_median(...)`: effettua le estrazioni dalla struttura dati, calcola il tempo per effettuare tutte le estrazioni e ne restituisce il valore

Alcune funzioni hanno al posto degli argomenti in ingresso il simbolo speciale "..." ma questo ha lo scopo di indicare che la funzione riceve vari oggetti in ingresso che però ho ritenuto essere ridondanti per la descrizione dei metodi.

4 Descrizione dei test eseguiti e risultati

In questa sezione tratteremo in modo approfondito i test, andando ad analizzare grafici e tabelle che ci permetteranno di osservare le prestazioni delle strutture dati che sono state messe alla prova in occasioni diverse. La principale suddivisione dei test è dettata dalla grandezza dell'array in input, contenente i valori numerici che verranno inizialmente inseriti nella struttura dati e successivamente estratti secondo l'ordine di priorità. La grandezza dell'array in input determina direttamente il numero di elementi delle code di priorità, perciò le sottosezioni successive rispecchieranno questa suddivisione.

- **Code con 100 elementi**: in questo caso analizzeremo le prestazioni delle code di priorità suddivise per tipo di array in input, per tipo di struttura dati e per tipo di operazione, in ogni grafico sarà presente una sola funzione.
- **Code con 1000 elementi**: in questa situazione invece saranno presenti più funzioni all'interno dello stesso grafico per poter visualizzare più dettagliatamente le prestazioni. Inizialmente si confronteranno le code variando il tipo di array in input e successivamente si confronteranno gli array in input variando le code, il tutto suddiviso ovviamente per l'operazione di inserimento e estrazione.
- **Code con 10000 elementi**: in questo caso si confronteranno all'interno dello stesso grafico le operazioni di inserimento e estrazione svolte sulla stessa struttura dati, suddiviso per tipo di array in input.

4.1 Raccolta dei tempi

Prima dell'analisi dei grafici vediamo come sono stati raccolti i tempi delle operazioni. Innanzitutto esaminiamo come questo è stato fatto per i tempi utilizzati nella generazione dei grafici. Come si può notare dai Listing 1 e 2 si esegue un ciclo `for` per l'intera lunghezza dell'array in input (`my_numpy_array`), si prende il tempo prima di eseguire l'operazione e immediatamente dopo, con l'ausilio della funzione `timer()` della libreria `timeit`, e infine si registra la differenza dei tempi nella matrice degli inserimenti o delle estrazioni.

```
1     def perform_insertion(my_data_structure, my_numpy_array, insertion_times_matrix, i):
2         for j, value in enumerate(my_numpy_array):
3             start_time = timer()
4             my_data_structure.insert(value)
5             end_time = timer()
6             insertion_times_matrix[i, j] = end_time - start_time
7         return insertion_times_matrix
```

Listing 1: Metodo che esegue gli inserimenti e salva i tempi singoli nella matrice

```
1     def perform_extraction(my_data_structure, my_numpy_array, extraction_times_matrix,i):
2         for j in range(len(my_numpy_array)):
3             start_time = timer()
4             my_data_structure.extract_max()
5             end_time = timer()
6             extraction_times_matrix[i, j] = end_time - start_time
7         return extraction_times_matrix
```

Listing 2: Metodo che esegue le estrazioni e salva i tempi singoli nella matrice

Il Listig 3 mostra una parte di codice della funzione main, notiamo che dopo aver inizializzato le matrici, che sfruttano la libreria `NumPy`, si eseguono i test all'interno di un ciclo `for` che li ripete `NUMBER_OF_TEST` volte, quest'ultima è una costante con valore assegnato 50. Successivamente si calcola la media dei tempi dei singoli inserimenti e estrazioni, si ottiene così la media del tempo di inserimento del primo elemento, del secondo elemento e così via fino all'ultimo. Stessa cosa per le estrazioni. Ciò ci permette di avere dei valori più affidabili. Infine si crea il grafico grazie alle operazioni di `plot_operation_times()` che sfruttano le funzioni della libreria `matplotlib`.

```
    # Inizializza le matrici numpy
    insertion_times_matrix = np.zeros((NUMBER_OF_TEST, array_size.value))
    extraction_times_matrix = np.zeros((NUMBER_OF_TEST, array_size.value))

    # Si esegue i test
    for i in range(NUMBER_OF_TEST):
        # Misura il tempo di inserimento degli elementi nella struttura dati
        insertion_times_matrix = perform_insertion(my_data_structure, my_numpy_array,
        insertion_times_matrix, i)

        # Misura il tempo di estrazione degli elementi dalla struttura dati
        extraction_times_matrix = perform_extraction(my_data_structure, my_numpy_array,
        extraction_times_matrix, i)

    # Calcola le medie lungo le colonne
    average_insertion_times = np.mean(insertion_times_matrix, axis=0)
    average_extraction_times = np.mean(extraction_times_matrix, axis=0)

    # Crea il grafico finale per le medie dei tempi di inserimento
    plot_operation_times_small(range(1, array_size.value + 1), average_insertion_times,
    data_structure_type, array_type, "Inserimento")

    # Crea il grafico finale per le medie dei tempi di estrazione
    plot_operation_times_small(range(1, array_size.value + 1), average_extraction_times,
    data_structure_type, array_type, "Estrazione")
```

Listing 3: Snippet di codice del metodo main() in cui vengono usati i metodi perform_insertion() e perform_extraction()

Consideriamo adesso le funzioni che sono state utilizzare per generare i tempi medi e mediani, calcolati sul tempo necessario per inserire o estrarre *tutti* gli elementi da una struttura dati. Nei Listing 4 e 5 l'approccio è simile ai casi precedenti ma l'intento è diverso, perciò il tempo viene preso prima dell'inizio e dopo la fine del ciclo `for` e infine viene restituita la differenza.

```
    def perform_insertion_mean_median(my_data_structure, my_numpy_array):
        start_total_insertion_time = timer()
        for j, value in enumerate(my_numpy_array):
            my_data_structure.insert(value)
        end_total_insertion_time = timer()
        return end_total_insertion_time - start_total_insertion_time
```

Listing 4: Metodo che esegue gli inserimenti e restituisce il tempo totale

```
    def perform_extraction_mean_median(my_data_structure, my_numpy_array):
        start_total_extraction_time = timer()
        for j in range(len(my_numpy_array)):
            my_data_structure.extract_max()
        end_total_extraction_time = timer()
        return end_total_extraction_time - start_total_extraction_time
```

Listing 5: Metodo che esegue le estrazioni e restituisce il tempo totale

Nel Listing 6 viene mostrato l'utilizzo delle funzioni precedenti. Anche in questo caso si esegue i test `NUMBER_OF_TEST` volte, ogni tempo calcolato viene salvato all'interno di array `NumPy` e successivamente si calcola la media e la mediana dei valori calcolati.

```
    # Inizializza gli array numpy
    total_insertion_times_array = np.zeros(NUMBER_OF_TEST)
    total_extraction_times_array = np.zeros(NUMBER_OF_TEST)

    # Si esegue i test
    for i in range(NUMBER_OF_TEST):
        # Misura il tempo di inserimento di tutti gli elementi nella struttura dati
        total_insertion_times_array[i] = perform_insertion_mean_median(my_data_structure,
        my_numpy_array_random)

        # Misura il tempo di estrazione di tutti gli elementi dalla struttura dati
        total_extraction_times_array[i] = perform_extraction_mean_median(
        my_data_structure, my_numpy_array_random)

    # Calcolo del tempo medio per completare tutti gli inserimenti e tutte le estrazioni
    average_total_insertion_time = np.mean(total_insertion_times_array)
    average_total_extraction_time = np.mean(total_extraction_times_array)

    # Calcolo della mediana dei tempi totali di inserimento ed estrazione
    median_total_insertion_time = np.median(total_insertion_times_array)
    median_total_extraction_time = np.median(total_extraction_times_array)
```

Listing 6: Snippet di codice del metodo mean_and_median() in cui vengono usati i metodi perform_insertion_mean_median() e perform_extraction_mean_median()

4.2 Risultati dei test

Come detto precedentemente analizzeremo i risultati dei test suddivisi in base agli elementi inseriti e estratti dalle strutture dati, per ogni sezione visualizzeremo grafici disposti a gruppi, che ci permetteranno di rendere ancora più esplicito il confronto, e tabelle con valori medi e mediani dei tempi.
Per la rappresentazione dei grafici ho deciso di adottare la stessa scala per poter confrontare meglio le funzioni dei tempi, inoltre ho preferito, in alcuni casi, "perdere" i picchi assunti dalle funzioni a favore di una migliore definizione del resto dei punti, con l'idea che l'analisi dei grafici si concentri sull'andamento complessivo e non dei singoli casi.

4.2.1 Code con 100 elementi

Per l'operazione di inserimento:

- **Heap:** In questi grafici non si riesce ad apprezzare l'andamento logaritmico dei tempi, possiamo però notare che quando l'array in input è decrescente, in Figura 6, questo rappresenta il caso migliore. Ciò accade perché l'operazione di inserimento in questa particolare coda di priorità aggiunge il nuovo elemento in fondo all'array e controlla che la proprietà del max-heap sia rispettata, cioè il valore del nuovo elemento sia minore o uguale a quello del padre. Perciò inserendo un valore sempre più piccolo del precedente la proprietà non dovrà mai essere ripristinata. Sempre per lo stesso motivo possiamo notare che il caso opposto, quando l'array in input è crescente (Figura 5), rappresenta il caso peggiore e già con soli 100 elementi da inserire si riesce a intravedere l'andamento logaritmico. Infine l'input di tipo random, in Figura 4, rappresenta il caso medio e presenta maggiori variazioni temporali proprio perché il tempo di inserimento dipende dal valore dell'elemento appena inserito.

- **Lista concatenata:** In ogni occasione, indipendentemente dal tipo di array in input, ottiene i tempi migliori rispetto alle altre code, e già da questi primi grafici si può notare l'andamento costante dei tempi di inserimento.

- **Lista concatenata ordinata:** Questa coda ha un comportamento simmetrico rispetto all'heap, per quanto riguarda il caso migliore e peggiore. Come possiamo vedere in Figura 5, quando l'input è di tipo crescente, alla lista concatenata ordinata basterà inserire sempre in testa i nuovi elementi per mantenere l'ordinamento decrescente. Si comporta quindi come una semplice lista concatenata, infatti anche dai grafici possiamo vedere andamenti pressoché identici per questo specifico caso. Quando invece l'array in input è decrescente (Figura 6), la struttura dati dovrà scorrere tutta la lista per poter trovare la corretta posizione del nuovo elemento, infatti il grafico rispecchia un andamento lineare. Per quanto riguarda l'input random in Figura 4, si ottiene sempre un andamento dei tempi che tende ad essere lineare ma con molte oscillazioni poiché, come per l'heap, il tempo di inserimento del nuovo elemento dipende dal suo valore.

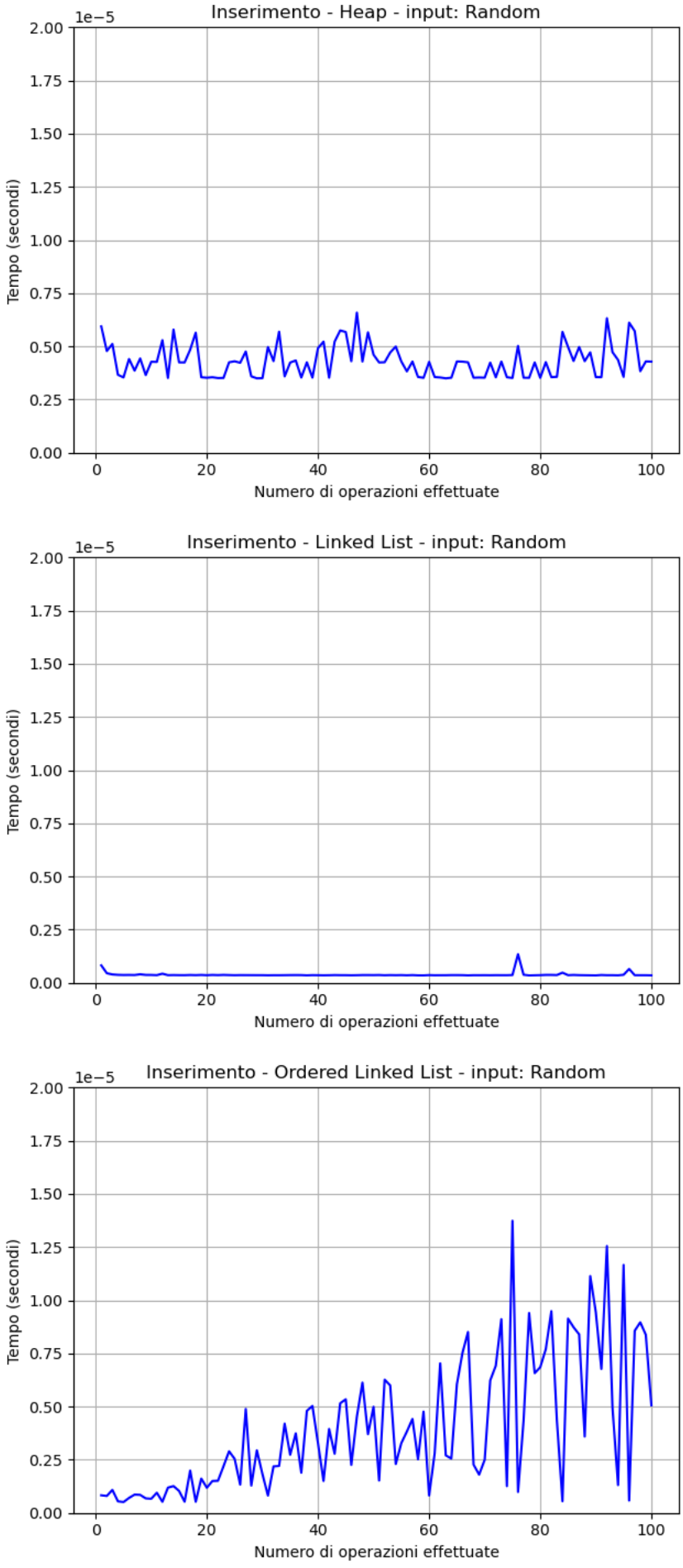

Figura 4: Inserimento di 100 elementi con array in input di tipo random

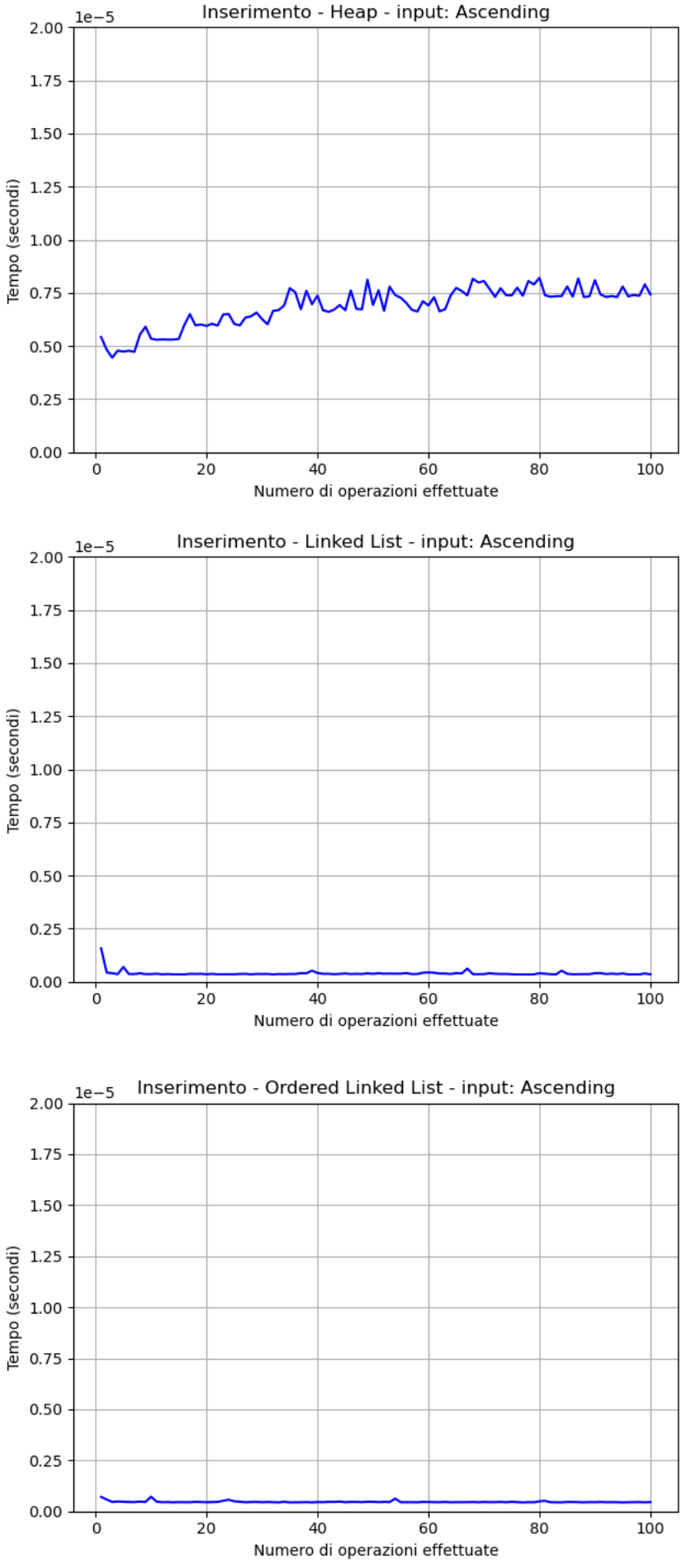

Figura 5: Inserimento di 100 elementi con array in input di tipo crescente

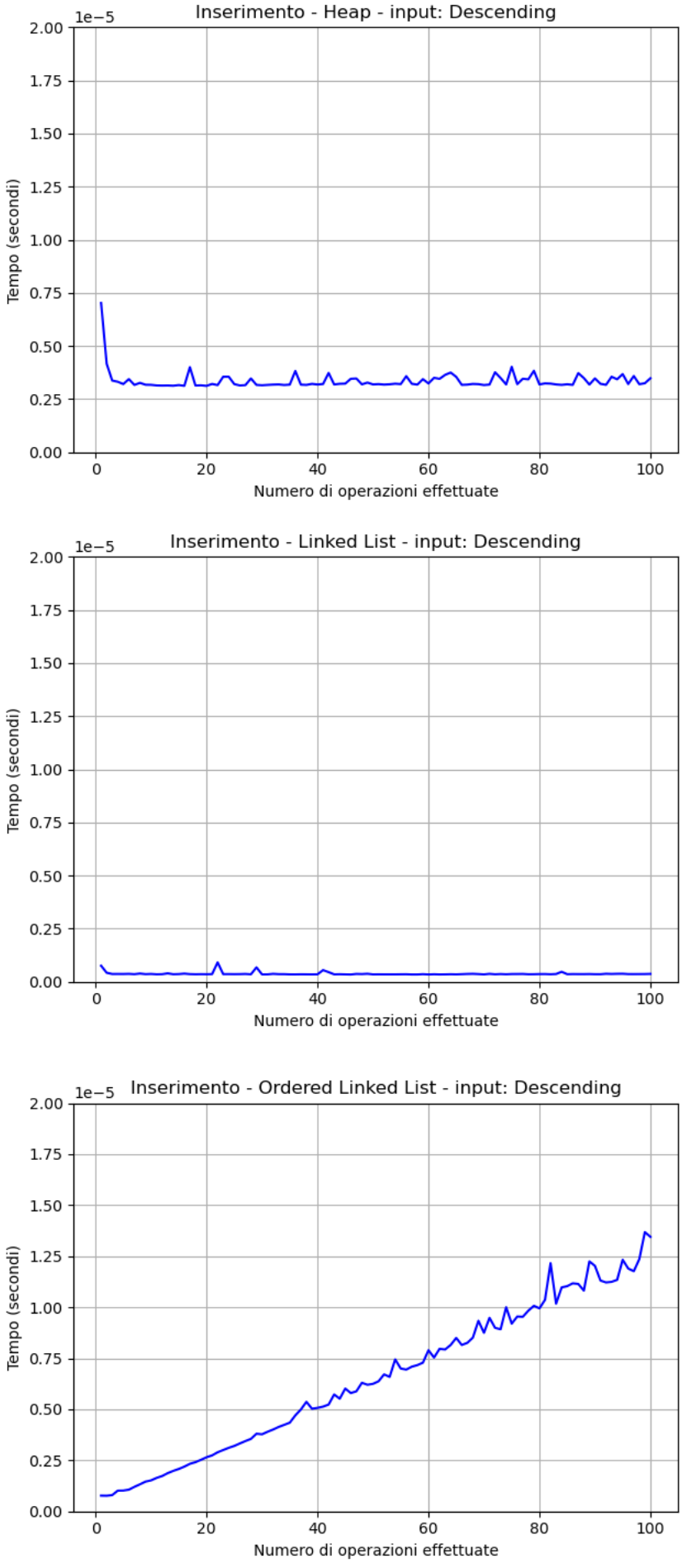

Figura 6: Inserimento di 100 elementi con array in input di tipo decrescente

Per l'operazione di estrazione:

- **Heap:** Nei grafici delle Figure 7, 8 e 9 si può apprezzare maggiormente l'andamento logaritmico rispetto all'operazione di inserimento. Indipendentemente dal tipo di input di ottengono grafici molto simili tra loro.
- **Lista concatenata:** Anche in questo caso i grafici non variano a seconda del tipo di array in input e si osserva un andamento lineare dei tempi di estrazione.
- **Lista concatenata ordinata:** È questa la struttura dati che ottiene i tempi migliori rispetto alle altre, per questo tipo di operazione, e li ottiene indipendentemente dal tipo di input.

Si nota un comportamento comune delle strutture dati, l'operazione di estrazione non è influenzata dal tipo di array in input, o per lo meno, non è influenzata tanto quanto lo è l'operazione di inserimento.

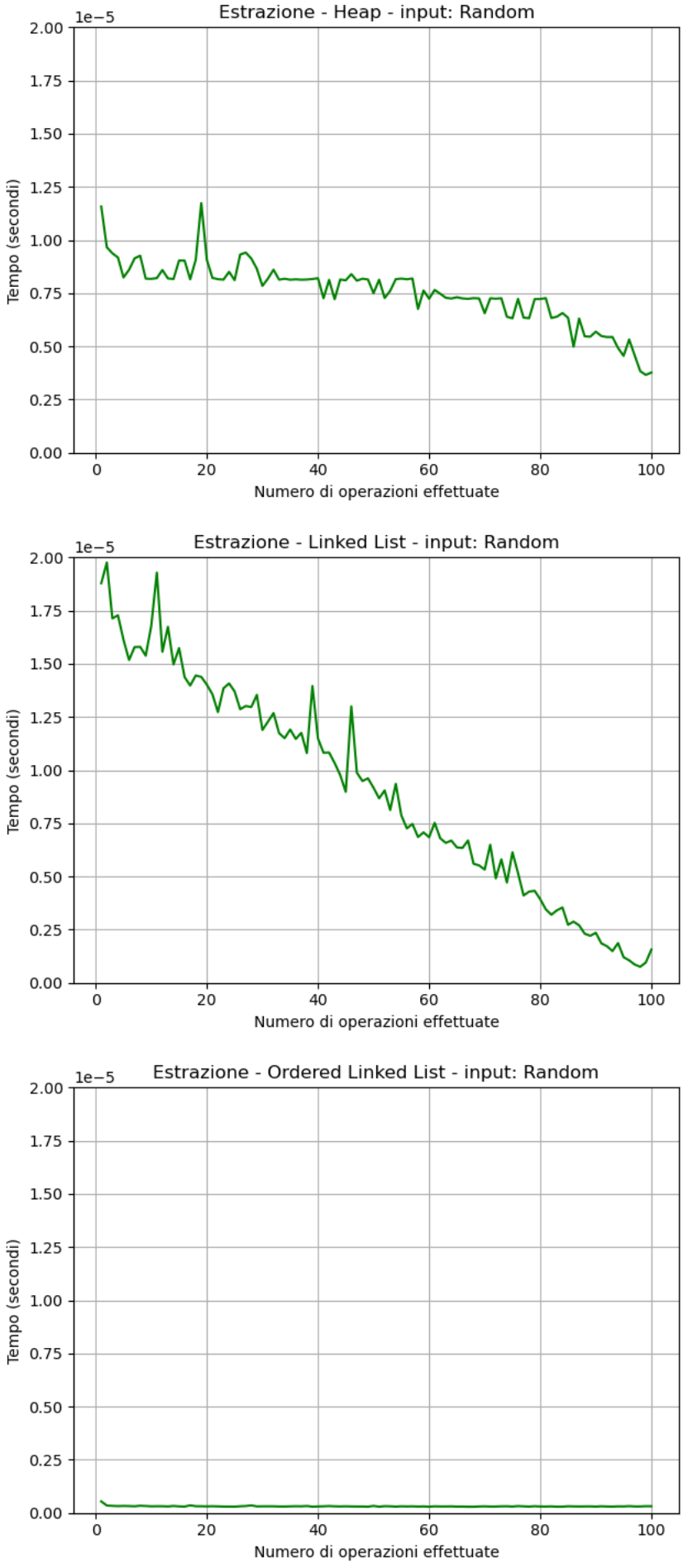

Figura 7: Estrazione di 100 elementi con array in input di tipo random

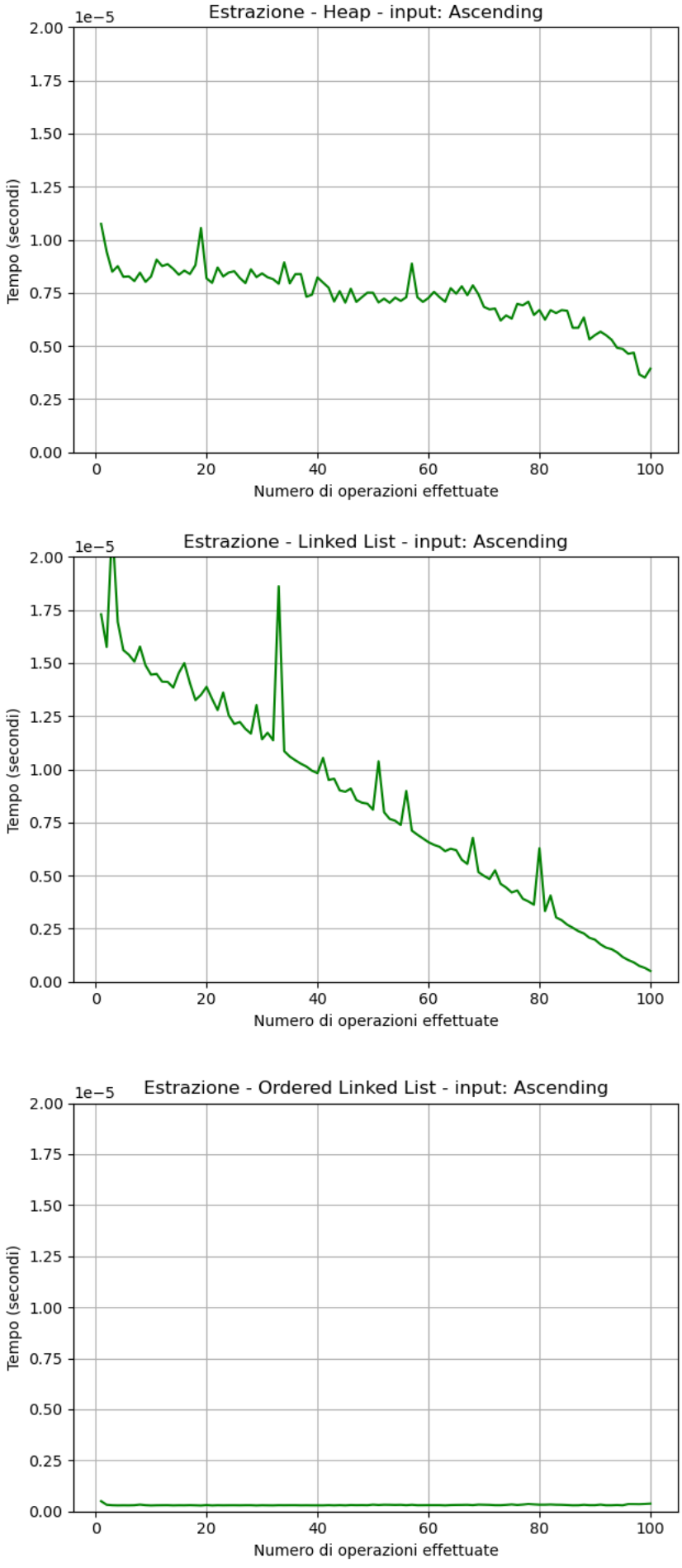

Figura 8: Estrazione di 100 elementi con array in input di tipo crescente

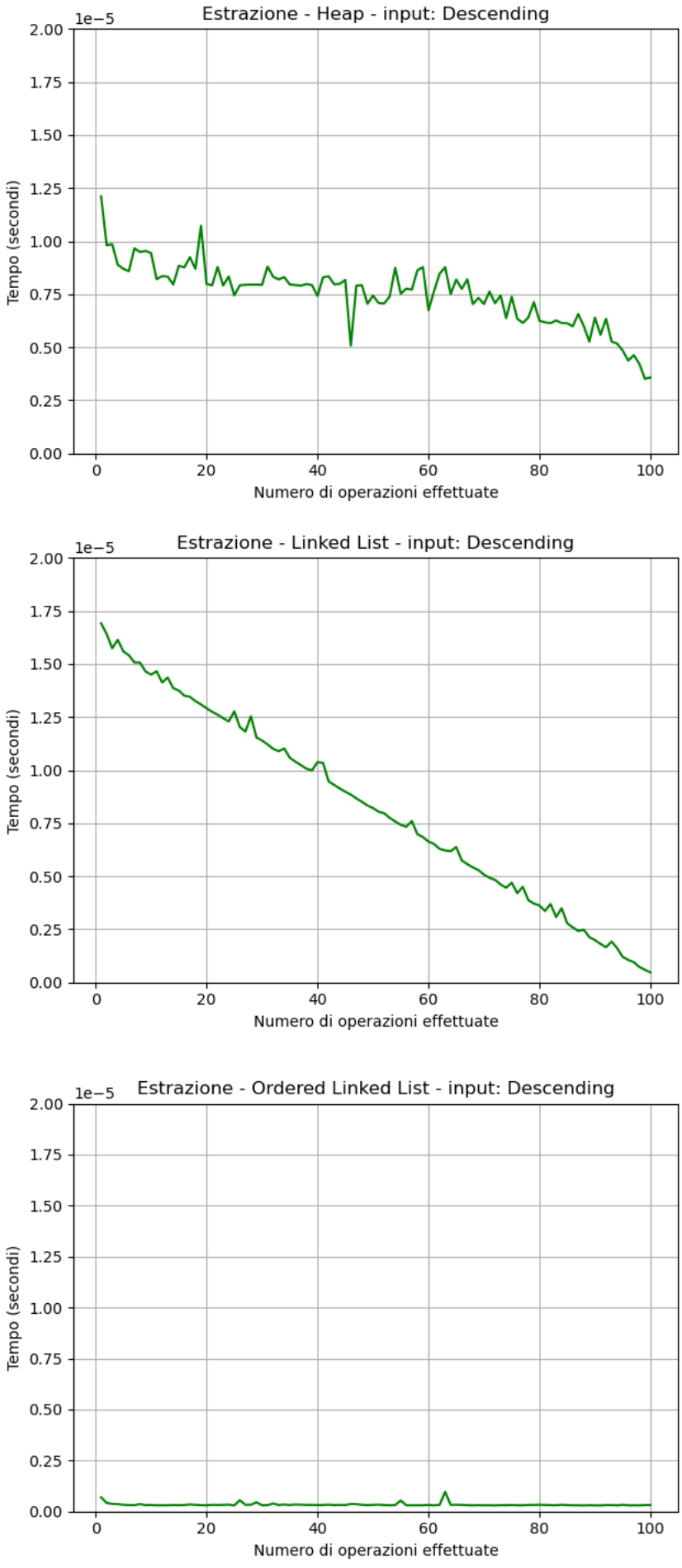

Figura 9: Estrazione di 100 elementi con array in input di tipo decrescente

Le tabelle seguenti rappresentano i tempi medi e mediani per inserire e estrarre tutti gli elementi nella e dalla struttura dati. I tempi sono espressi in secondi e facendo riferimento alla precedente analisi dei grafici sono stati evidenziati in verde i tempi che rappresentano i casi migliori e in rosso i casi peggiori. In generale tutti i valori rispecchiano l'analisi dei grafici, compresi i casi migliori, peggiori e gli andamenti costanti. Si nota che anche in questo caso i tempi calcolati sulle estrazioni non variano molto a seconda del tipo di input. Infine si osserva che la media e la mediana, prese nei singoli casi, sono pressoché identiche (piccole variazioni solo dopo i millesimi di secondo) e ciò ci indica che i risultati dei test sono stati molto simili tra loro, ci sono pochi casi limite e che la ripetizione dei test ha ammortizzato l'influenza di quest'ultimi.

Heap				
	Inserimento		Estrazione	
Random	Media: 0,000442	Mediana: 0,000423	Media: 0,000768	Mediana: 0,000742
Crescente	Media: 0,000690	Mediana: 0,000668	Media: 0,000731	Mediana: 0,000714
Decrescente	Media: 0,000327	Mediana: 0,000326	Media: 0,000743	Mediana: 0,000740

Tabella 2: Tempi medi e mediani per la struttura dati: Heap, con 100 operazioni

Lista concatenata				
	Inserimento		Estrazione	
Random	Media: 0,000034	Mediana: 0,000033	Media: 0,000776	Mediana: 0,000773
Crescente	Media: 0,000033	Mediana: 0,000032	Media: 0,000772	Mediana: 0,000761
Decrescente	Media: 0,000037	Mediana: 0,000035	Media: 0,000816	Mediana: 0,000810

Tabella 3: Tempi medi e mediani per la struttura dati: Lista concatenata, con 100 operazioni

Lista concatenata ordinata				
	Inserimento		Estrazione	
Random	Media: 0,000339	Mediana: 0,000331	Media: 0,000023	Mediana: 0,000022
Crescente	Media: 0,000043	Mediana: 0,000041	Media: 0,000023	Mediana: 0,000022
Decrescente	Media: 0,000639	Mediana: 0,000623	Media: 0,000025	Mediana: 0,000023

Tabella 4: Tempi medi e mediani per la struttura dati: Lista concatenata ordinata, con 100 operazioni

4.2.2 Code con 1000 elementi

Nelle Figure 10 e 11 possiamo vedere che i grafici rispecchiano i comportamenti analizzati precedentemente, quando le operazioni erano solo 100. Nell'operazione di inserimento si confermano i casi migliori e peggiori sia per l'heap che per la lista concatenata ordinata. Per l'heap notiamo che il caso medio, cioè quando l'array in input è di tipo random, si ha delle prestazioni molto simili al caso migliore. In ogni caso la differenza di tempo tra il caso migliore e quello peggiore è lieve. Per la lista concatenata ordinata è ben marcata la differenza di prestazioni a seconda del tipo di input, ad ogni modo è chiaro che quando l'input è random o decrescente l'andamento è lineare, invece se l'array in input è crescente si ha una complessità temporale costante. Per la lista concatenata non c'è niente di nuovo da mettere in mostra, come nel caso precedente si ottiene sempre una complessità costante indipendentemente dal tipo di input.

Anche in questo caso per l'operazione di estrazione si osserva che l'andamento dei tempi non è influenzato dal tipo di array in input. Inoltre notiamo, con un elevato grado di certezza, che la complessità temporale di questa operazione per l'heap è logaritmico, per la lista concatenata è lineare e per la lista concatenata ordinata è costante.

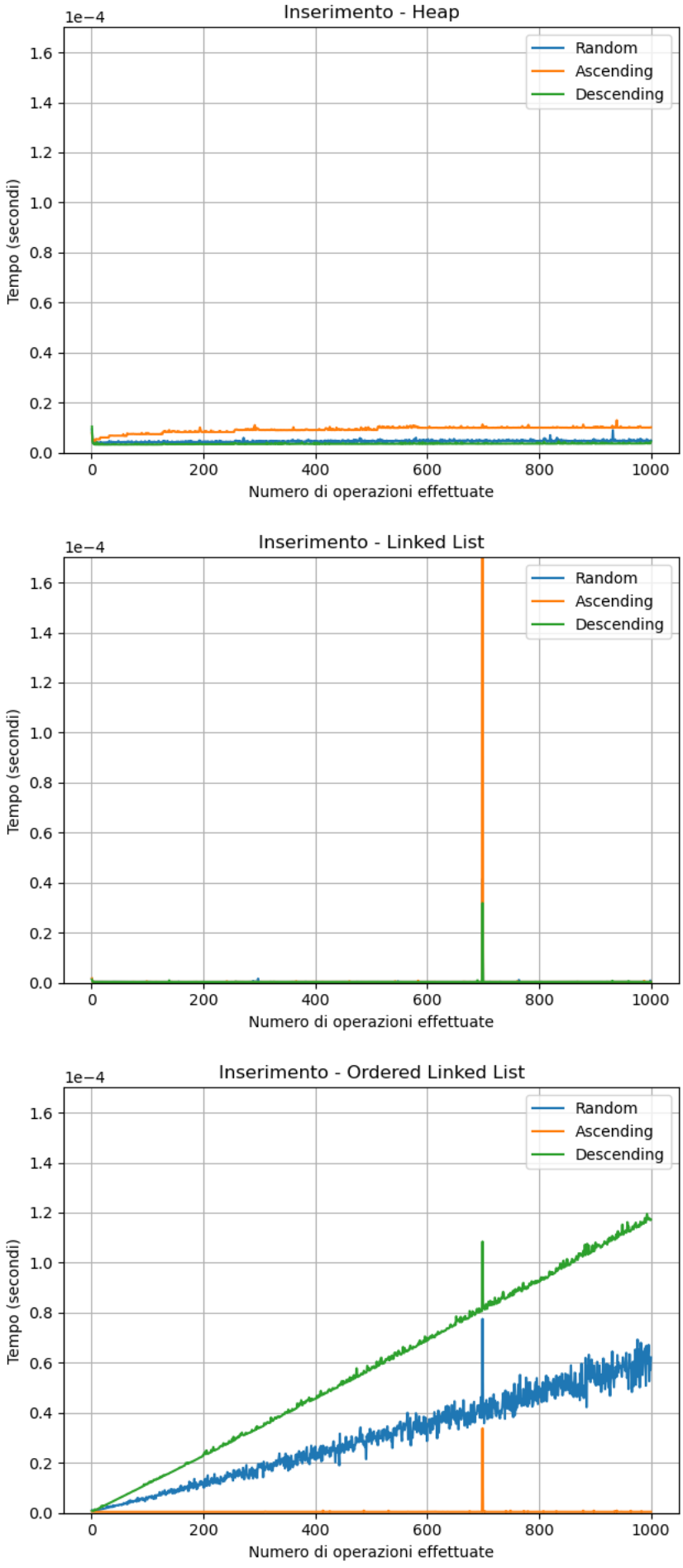

Figura 10: Inserimento di 1000 elementi nelle tre strutture dati variando il tipo di array in input

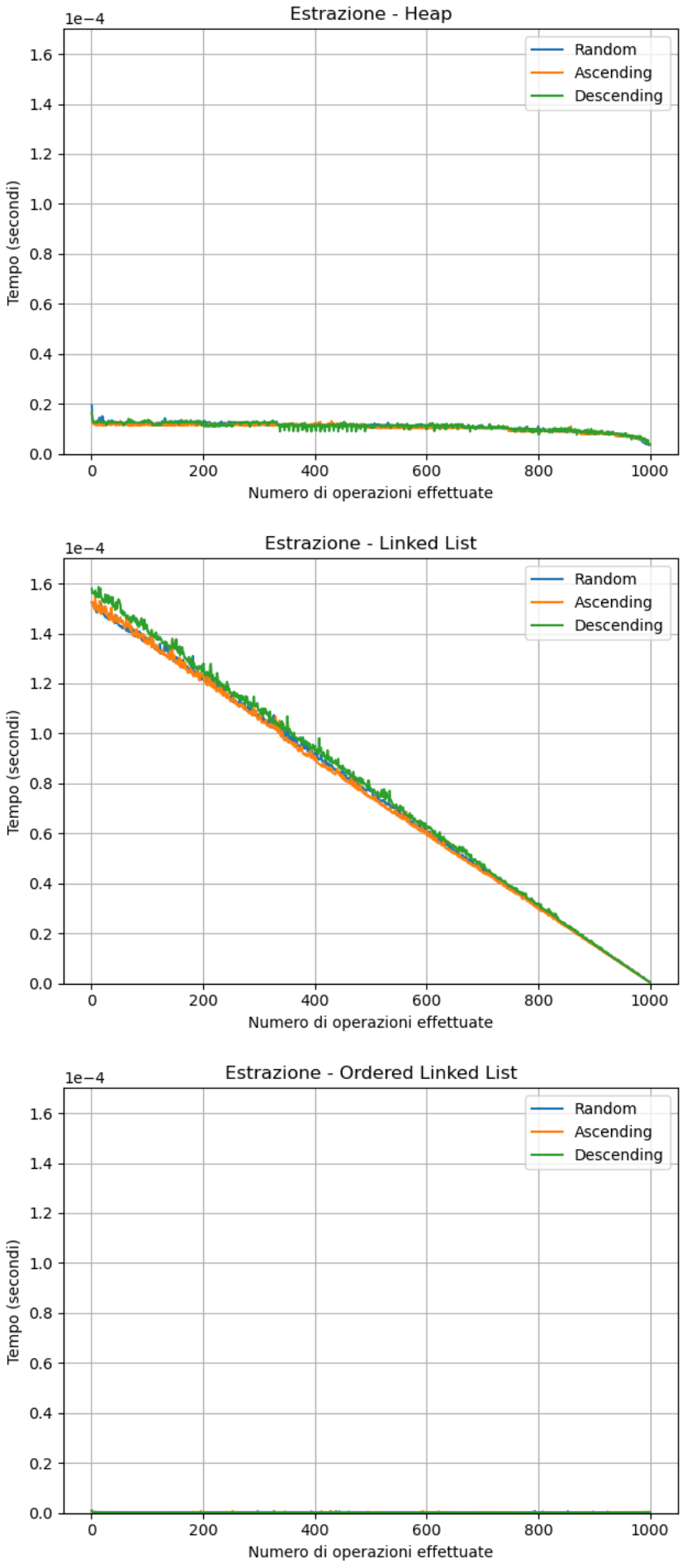

Figura 11: Estrazione di 1000 elementi nelle tre strutture dati variando il tipo di array in input

Nelle seguenti Figure, 12 e 13, confrontiamo all'interno dello stesso grafico le tre code di priorità a seconda del tipo di array in input. Nell'operazione di inserimento notiamo che quando l'input è random la lista concatenata ordinata ha le prestazioni peggiori, nonostante oscilli molto ha comunque un andamento tendente al lineare. Inoltre, di particolar importanza, si osserva che la differenza di prestazione tra l'heap e la lista concatenata è lieve. Considerazioni simili anche quando il tipo di input è decrescente, la principale differenza rispetto all'input random è che la complessità temporale della lista concatenata ordinata ha minori oscillazioni perché ogni inserimento rappresenta il suo caso peggiore, perciò i tempi tendono sempre ad aumentare. Quando l'array in input è crescente la lista concatenata ordinata si comporta come una semplice lista concatenata, come già detto nei paragrafi precedenti, quindi l'andamento dell'heap è il peggiore dei tre. Nonostante sia anche il suo caso peggiore la differenza in termini assoluti non è ampia.

Questo tipo di analisi, rende ancor più evidente il fatto che l'array in input non influenza le prestazioni dell'operazione di estrazione, poiché sono stati generati tre grafici praticamente uguali. Come nei casi precedenti possiamo apprezzare esplicitamente l'andamento della complessità temporale che caratterizza le strutture dati prese in esame.

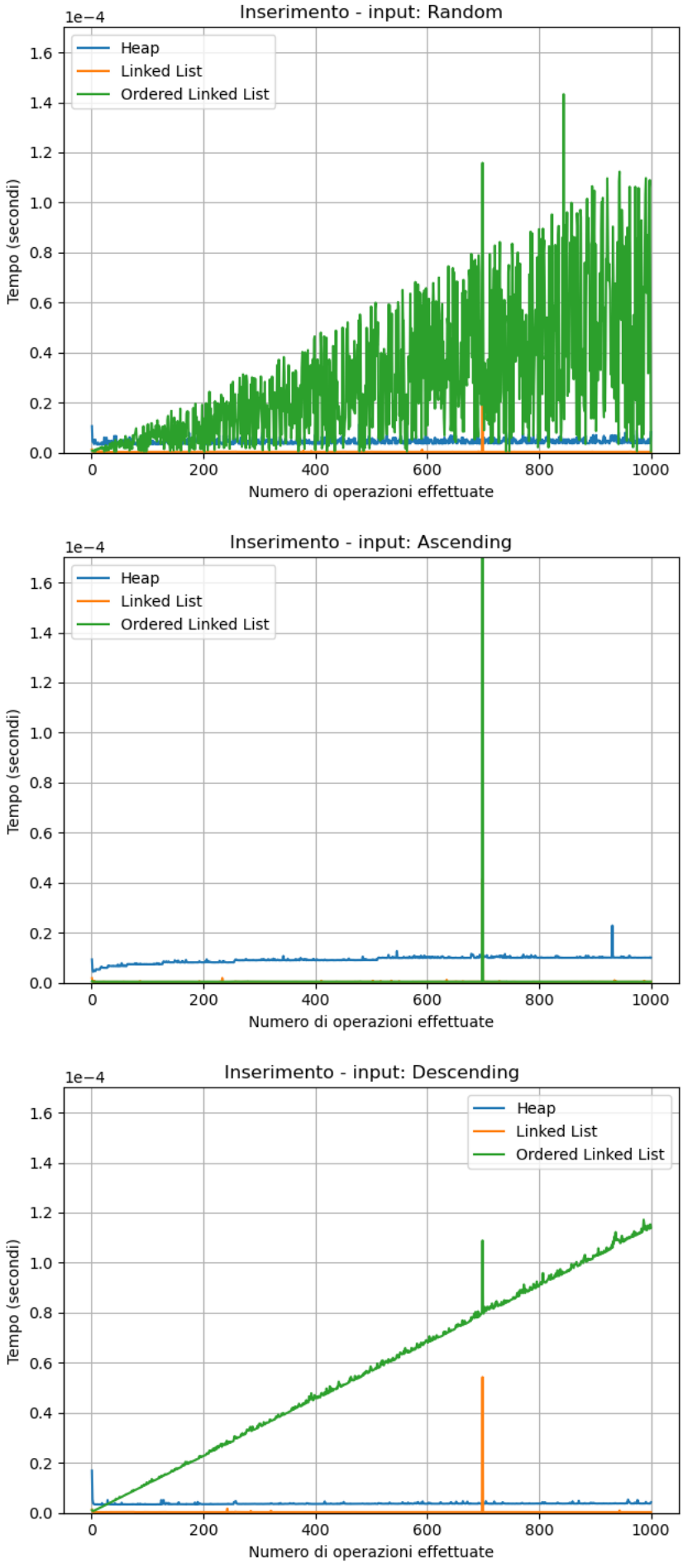

Figura 12: Inserimento di 1000 elementi per tipo di array in input variando le strutture dati

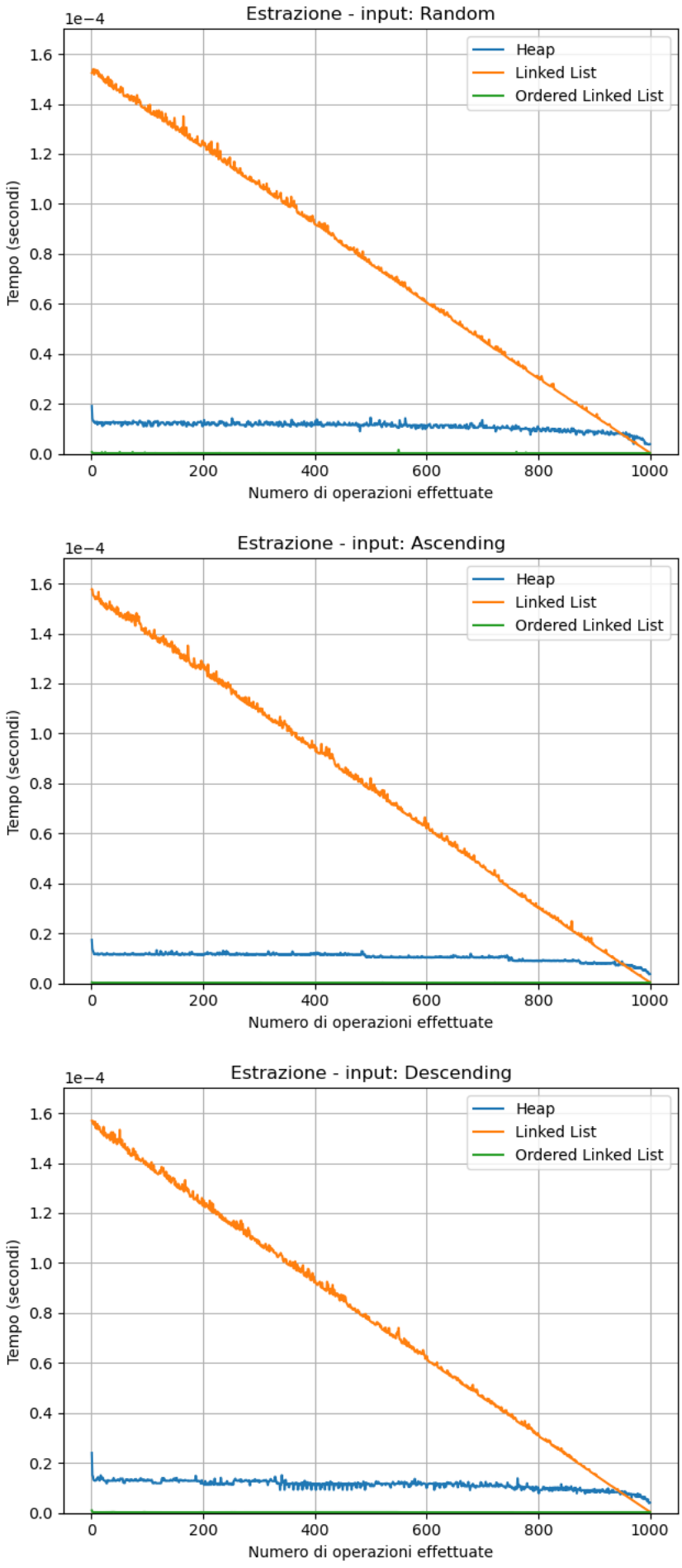

Figura 13: Estrazione di 1000 elementi per tipo di array in input variando le strutture dati

Anche in questa sezione le seguenti tabelle rappresentano i tempi medi e mediani per eseguire tutte le operazioni sulle strutture dati. I tempi sono sempre espressi in secondi, in verde sono evidenziati i tempi che rappresentano i casi migliori per l'heap e per la lista concatenata ordinata, in rosso i casi peggiori, sempre per queste due code. In generale, come per le code con 100 elementi, media e mediana sono pressoché uguali presi i singoli casi, tranne per il caso dell'inserimento nella lista concatenata quando l'array in input è decrescente, caso evidenziato in giallo. La media è superiore alla mediana, andando a vedere nel codice l'array che contiene i tempi totali di questo specifico caso (quello su cui poi si calcola la media e la mediana) si nota che, su più esecuzioni, la 15^a volta che viene effettuato il test si ottiene un valore circa 30 volte più grande rispetto agli altri test. Ciò spiega perché la media è più grande rispetto alla mediana, penso però che il valore più significativo da considerare sia la mediana, simile anche agli altri tempi di inserimento nella lista concatenata.

Heap				
	Inserimento		Estrazione	
Random	Media: 0,004426	Mediana: 0,004390	Media: 0,010820	Mediana: 0,010802
Crescente	Media: 0,009038	Mediana: 0,008958	Media: 0,010465	Mediana: 0,010386
Decrescente	Media: 0,003474	Mediana: 0,003480	Media: 0,010798	Mediana: 0,010835

Tabella 5: Tempi medi e mediani per la struttura dati: Heap, con 1000 operazioni

Lista concatenta				
	Inserimento		Estrazione	
Random	Media: 0,000373	Mediana: 0,000366	Media: 0,075531	Mediana: 0,075527
Crescente	Media: 0,000365	Mediana: 0,000360	Media: 0,074490	Mediana: 0,074504
Decrescente	Media: 0,000584	Mediana: 0,000364	Media: 0,078060	Mediana: 0,077918

Tabella 6: Tempi medi e mediani per la struttura dati: Lista concatenata, con 1000 operazioni

Lista concatenata ordinata				
	Inserimento		Estrazione	
Random	Media: 0,027654	Mediana: 0,027654	Media: 0,000242	Mediana: 0,000239
Crescente	Media: 0,000446	Mediana: 0,000443	Media: 0,000233	Mediana: 0,000232
Decrescente	Media: 0,057808	Mediana: 0,057800	Media: 0,000238	Mediana: 0,000233

Tabella 7: Tempi medi e mediani per la struttura dati: Lista concatenata ordinata, con 1000 operazioni

4.2.3 Code con 10000 elementi

Quest'ultimo caso di test ci conferma ciò che abbiamo dedotto dai casi precedenti. Per la struttura dati heap in Figura 14, si nota che l'andamento dei tempi, sia per l'operazione di inserimento che per quella di estrazione, è logaritmico. Anche in questo caso, per l'inserimento, l'input decrescente rappresenta il caso migliore, random il caso medio (comunque paragonabile al caso migliore), e l'input crescente rappresenta il caso peggiore. Quando si tratta di estrazione gli andamenti delle funzioni temporali sono paragonabili indipendentemente dal tipo di array in input e si apprezza ancor di più la curva logaritmica dei tempi. Per la lista concatenata, Figura 15, si ottiene sempre un andamento costate per l'operazione di inserimento, e un andamento lineare per l'estrazione. Infine la lista concatenata ordinata in Figura 16, per l'operazione di inserimento, ha un andamento lineare sia quando l'input è random che quando è decrescente, nel primo caso però sono presenti molte oscillazioni che invece non ci sono nel secondo. Quando l'input è crescente la complessità temporale dell'operazione di inserimento è costante. Per quanto riguarda l'estrazione si ha sempre un andamento costante in tutti e tre i grafici.

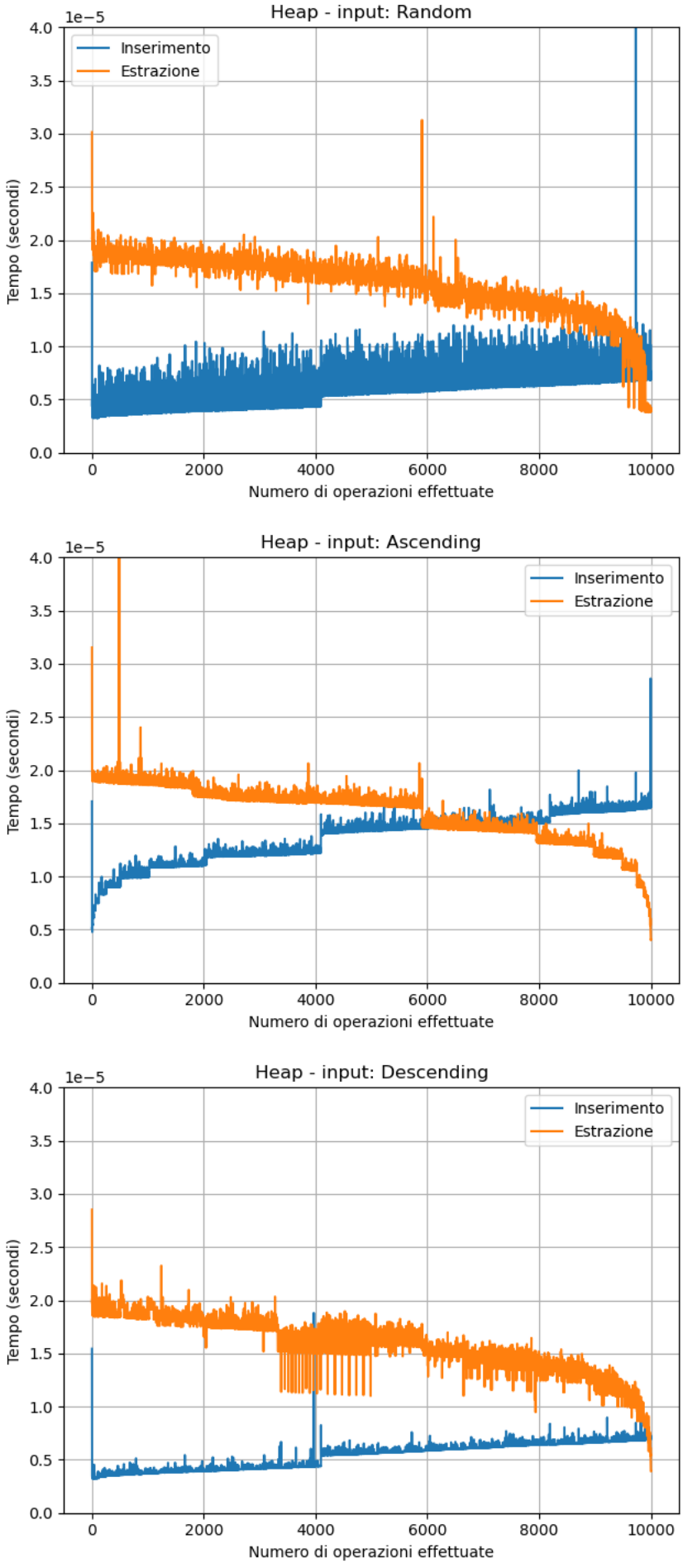

Figura 14: Inserimento e estrazione di 10000 elementi nell'heap variando il tipo di input

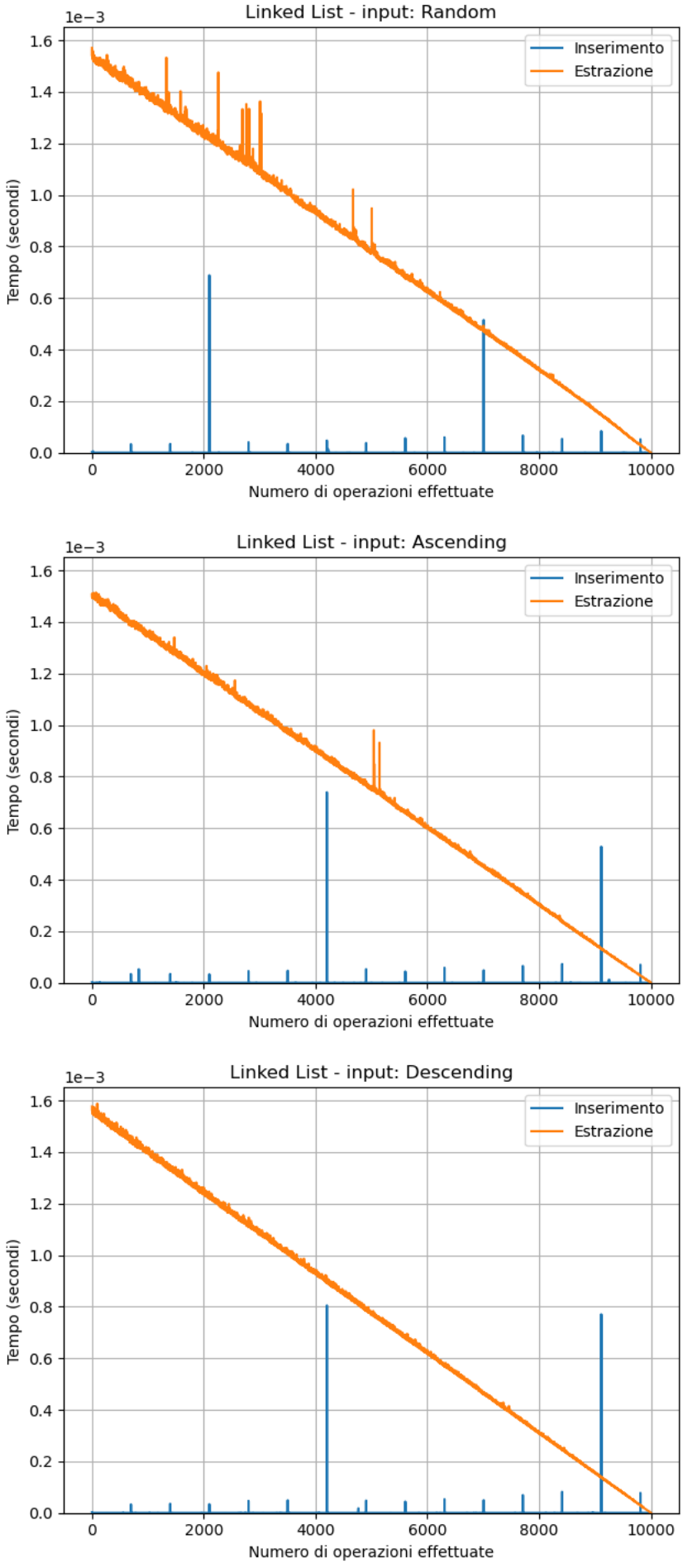

Figura 15: Inserimento e estrazione di 10000 elementi nella lista concatenata variando il tipo di input

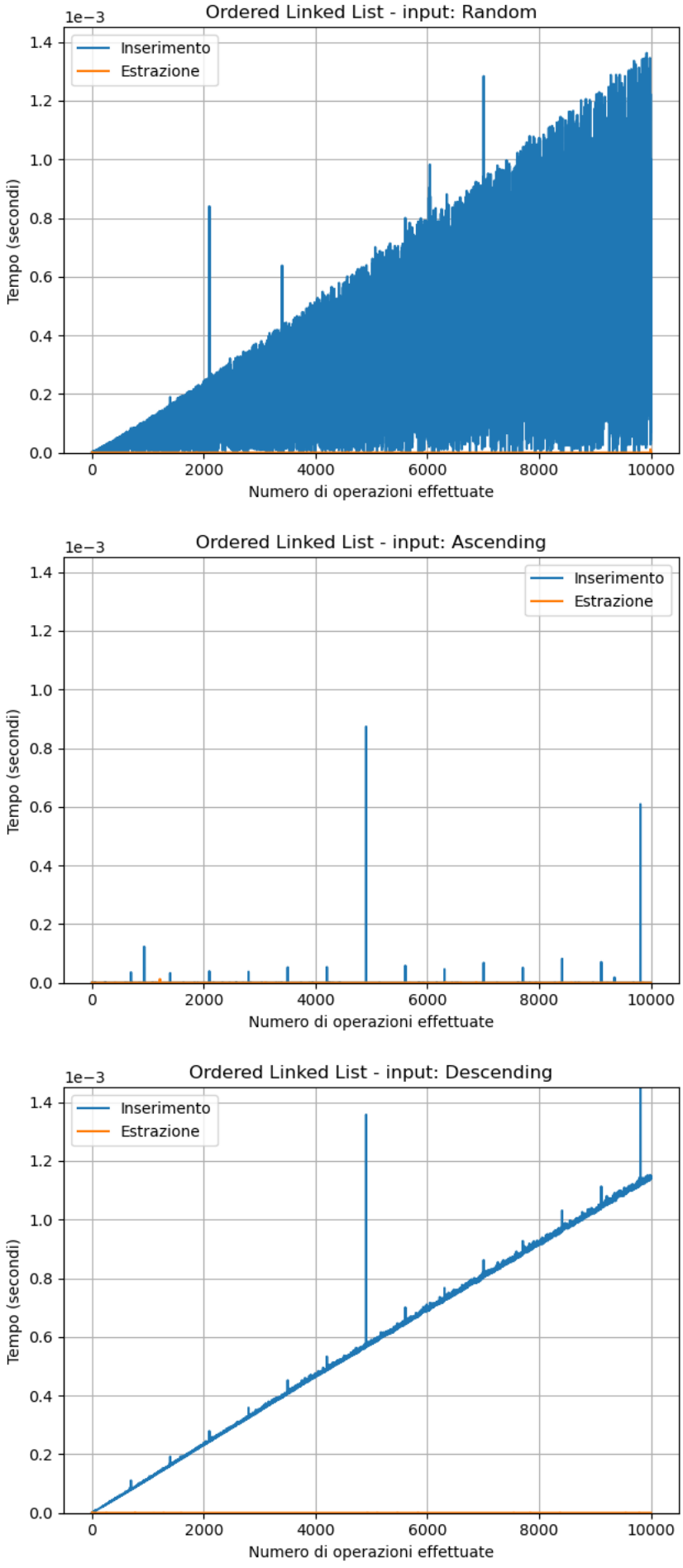

Figura 16: Inserimento e estrazione di 10000 elementi nella lista concatenata ordinata variando il tipo di input

Infine le tabelle dei tempi medi e mediani con 10000 operazioni, i tempi sono espressi in secondi, in verde i casi migliori per heap e lista concatenata, in rosso i casi peggiori, sempre per queste due strutture dati. Per la struttura dati heap notiamo che il caso medio, cioè l'input random, è leggermente peggiore del caso migliore e si avvicina molto a quest'ultimo in termini di prestazioni. Il caso peggiore si riconferma con l'input di tipo crescente. I tempi di estrazione sono peggiori dei tempi di inserimento ma sempre uguali indipendentemente dal tipo di input. Per la lista concatenata notiamo che la media dei tempi di inserimento è sempre maggiore della mediana, fenomeno simile al caso con 1000 elementi quando l'array in input è decrescente, ma stavolta accade con ogni tipo di input, e andando a vedere l'array che contiene i tempi totali notiamo che questa volta ci sono alcuni picchi durante l'esecuzione dei test, per questo la media è più grande. Ad ogni modo la maggior parte dei tempi è simile al valore mediano perciò reputo questo dato sia il più significativo. Media e mediana per l'operazione di estrazione invece sono sempre molto simili sia nei casi specifici che variando il tipo di input. Per la lista concatenata ordinata, nell'operazione di inserimento, a seconda dell'array in input i tempi variano molto, persino di secondi, perciò notiamo una gran differenza tra caso migliore, medio e peggiore. Sull'operazione di estrazione invece si ottengono valori molto simili, sia tra media e mediana, che variando il tipo di input.

Heap				
	Inserimento		Estrazione	
Random	Media: 0,063609	Mediana: 0,063406	Media: 0,158566	Mediana: 0,158793
Crescente	Media: 0,135575	Mediana: 0,135410	Media: 0,156544	Mediana: 0,156644
Decrescente	Media: 0,052839	Mediana: 0,052843	Media: 0,159295	Mediana: 0,159491

Tabella 8: Tempi medi e mediani per la struttura dati: Heap, con 10000 operazioni

Lista concatenata				
	Inserimento		Estrazione	
Random	Media: 0,005368	Mediana: 0,004222	Media: 7,724123	Mediana: 7,755459
Crescente	Media: 0,005201	Mediana: 0,004093	Media: 7,699711	Mediana: 7,684440
Decrescente	Media: 0,005174	Mediana: 0,004075	Media: 7,822481	Mediana: 7,819284

Tabella 9: Tempi medi e mediani per la struttura dati: Lista concatenata, con 10000 operazioni

Lista concatenata ordinata				
	Inserimento		Estrazione	
Random	Media: 3,355563	Mediana: 3,355921	Media: 0,002651	Mediana: 0,002634
Crescente	Media: 0,006399	Mediana: 0,005077	Media: 0,002766	Mediana: 0,002419
Decrescente	Media: 5,764218	Mediana: 5,768061	Media: 0,002348	Mediana: 0,002348

Tabella 10: Tempi medi e mediani per la struttura dati: Lista concatenata ordinata, con 10000 operazioni

5 Riflessioni finali

L'analisi dei risultati ottenuti dai test ha evidenziato i seguenti esiti:

- La struttura dati heap è l'unica che non ha mai un costo computazionale costante e lineare ma sempre logaritmico, da un lato perciò non riesce ad affermasi come la più efficiente nei singoli casi, dall'altro non è quasi mai la peggiore (solo quando si tratta inserimento con input crescente è la peggiore tra le strutture dati). Nonostante ciò le prestazioni logaritmiche non si distanziano molto dalle prestazioni costanti in termini di efficienza, sopratutto se il numero di operazioni è elevato. Tutto ciò, rende questa coda la più versatile e la più efficiente, in particolar modo se le operazioni di inserimento e estrazione avvengono con la stessa frequenza.
- La lista concatenata ottiene sempre le migliori prestazioni quando si tratta di inserimento, ma le peggiori quando si effettuano le estrazioni. A differenza delle altre code non presenta mai caso peggiore, migliore o medio, indipendentemente dal tipo di input ha sempre tempi computazionali costanti per l'inserimento e lineari per l'estrazione.
- La lista concatenata ordinata ha una complessità lineare per l'operazione di inserimento, tranne per il suo caso migliore, quando il tipo di array in input è crescente, e si comporta come una semplice lista concatenata. Per l'operazione di estrazione di afferma come la miglior struttura dati da un punto di vista computazionale, poiché ha sempre una complessità costante.
- Tutti i test eseguiti e l'analisi derivata da questi è in completo accorto con i risultati teorici del libro del corso.

Riferimenti bibliografici

[1] T.H. Cormen et al. *Introduzione agli algoritmi e strutture dati*. Collana di istruzione scientifica. Serie di informatica. McGraw-Hill Companies, 2010. ISBN: 9788838665158. URL: https://books.google.it/books?id=5gopSgAACAAJ.