



UNIVERSITÀ  
DEGLI STUDI  
FIRENZE

UNIVERSITÀ DEGLI STUDI DI FIRENZE  
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

---

## Laboratorio di Algoritmi e Strutture Dati

---

*Autore:*  
Andrea Pistelli

*Corso principale:*  
Algoritmi e Strutture Dati

*N° Matricola:*  
7049769

*Docente corso:*  
Simone Marinai

## Indice

<b>1</b>	<b>Introduzione</b>	<b>3</b>
1.1	Descrizione del progetto . . . . .	3
1.2	Specifiche della piattaforma di test . . . . .	3
<b>2</b>	<b>Analisi teorica del problema</b>	<b>4</b>
2.1	Code di priorità . . . . .	4
2.2	Operazioni fondamentali . . . . .	4
2.3	Implementazioni nelle diverse strutture dati . . . . .	4
2.4	Prestazioni attese . . . . .	4
<b>3</b>	<b>Documentazione del codice</b>	<b>5</b>
3.1	Schema delle classi e scelte implementative . . . . .	5
3.2	Descrizione dei metodi principali di ogni classe o modulo . . . . .	6
3.3	Note sull'implementazione . . . . .	7
<b>4</b>	<b>Descrizione e analisi del primo esperimento - Comparazione di performance</b>	<b>8</b>
4.1	Dati utilizzati . . . . .	8
4.2	Misurazioni . . . . .	8
4.3	Risultati sperimentali . . . . .	8
4.3.1	valori casuali . . . . .	8
4.3.2	valori ascendenti . . . . .	9
4.3.3	valori discendenti . . . . .	11
4.4	Analisi risultati . . . . .	12
<b>5</b>	<b>Descrizione e analisi del secondo esperimento - Analisi asintotica</b>	<b>13</b>
5.1	Dati utilizzati . . . . .	13
5.2	Misurazioni . . . . .	13
5.3	Risultati sperimentali . . . . .	14
5.3.1	Max Heap . . . . .	14
5.3.2	Lista non ordinata . . . . .	15
5.3.3	Lista ordinata . . . . .	16
5.4	Analisi risultati . . . . .	17
<b>6</b>	<b>Conclusioni</b>	<b>18</b>

## Elenco delle figure

1	Complessità attese . . . . .	4
2	Diagramma delle classi e dei moduli . . . . .	5
3	casuali inserimento . . . . .	8
4	casuali estrazioni . . . . .	8
5	casuali visualizzazione massimo . . . . .	9
6	casuali misto . . . . .	9
7	ascendenti inserimento . . . . .	9
8	ascendenti estrazione . . . . .	10
9	ascendenti visualizzazione massimo . . . . .	10
10	ascendenti misto . . . . .	10
11	discendenti inserimento . . . . .	11
12	discendenti estrazione . . . . .	11
13	discendenti visualizzazione massimo . . . . .	11
14	discendenti misto . . . . .	12
15	Max Heap Inserimento . . . . .	14
16	Max Heap estrazione . . . . .	14
17	Max Heap visualizzazione massimo . . . . .	14
18	Lista non ordinata Inserimento . . . . .	15
19	Lista non ordinata estrazione . . . . .	15
20	Lista non ordinata visualizzazione massimo . . . . .	15
21	Lista ordinata Inserimento . . . . .	16
22	Lista ordinata estrazione . . . . .	16

23	Lista ordinata visualizzazione massimo	16
----	--	----

# 1 Introduzione

## 1.1 Descrizione del progetto

La seguente relazione si pone l'obiettivo assegnatomi di analizzare le differenze tra le seguenti implementazioni tra le code di priorità

- Heap
- Lista Concatenata
- Lista Concatenata Ordinata

Per questa analisi è stato quindi necessario scrivere programmi in Python che implementassero quanto richiesto e ci permettessero di osservare le differenze.

## 1.2 Specifiche della piattaforma di test

I test verranno eseguiti sullo stesso dispositivo che ha le seguenti caratteristiche:

- **CPU** : AMD Ryzen 7 5700U with Radeon G
- **GPU** : AMD ATI 05:00.0 Lucienne
- **RAM** : 16 GB
- **OS** : Ubuntu 22.04.5 LTS 16bit

La piattaforma in cui il codice è stato scritto e eseguito è l'IDE **PyCharm Professional 2024.1.3**. La stesura di questo testo è avvenuta tramite l'utilizzo dell'editor online **Overleaf**.

## 2 Analisi teorica del problema

### 2.1 Code di priorità

Le code di priorità sono strutture dati che gestiscono elementi ordinandoli secondo una priorità assegnata, invece di utilizzare il classico ordine FIFO (First In, First Out) delle code normali. In una coda di priorità, ogni elemento ha un valore di priorità, e l'elemento con la priorità più alta viene servito per primo. Questo tipo di coda è particolarmente utile in scenari come la gestione dei processi di un sistema operativo, dove alcuni compiti devono avere la precedenza su altri, o negli algoritmi di ricerca come Dijkstra, dove vengono gestiti percorsi ottimali.

### 2.2 Operazioni fondamentali

Tre operazioni sono fondamentali alle le Code di Priorità per svolgere il loro compito:

- **Inserimento:** Le code devono potere inserire un oggetto con una determinata priorità nella lista
- **Estrazione:** Le code devono poter velocemente estrarre l'elemento con priorità maggiore dalla lista
- **Trova massimo:** più comunemente chiamata peek in inglese, serve a trovare e restituire l'elemento massimo senza eliminarlo dalla coda

### 2.3 Implementazioni nelle diverse strutture dati

- **Heap**

La coda viene implementata con un heap binario. L'inserimento è implementato inserendo l'elemento nell'ultima posizione e chiamando la funzione *heapify\_up* che iterativamente scorre l'albero dalla foglia alla radice, o fino a che non deve più fare modifiche, per mantenere le proprietà di heap binario. Il peek restituisce il primo valore della lista e l'estrazione restituisce il primo valore della lista e lo cancella, poi chiama la funzione *heapify\_down* che iterativamente scorre la lista dalla radice ad una foglia per riempire il nodo vuoto creato in modo da mantenere la proprietà di heap binario.

- **Lista concatenata non ordinata**

Nella lista non ordinata l'inserimento è implementato semplicemente aggiungendo in coda l'elemento. L'estrazione e il peek sono implementati facendo una ricerca in lunghezza sulla lista.

- **Lista concatenata ordinata**

Nella lista ordinata l'inserimento è implementato scorrendo la lista fino a alla posizione corretta per mantenere l'ordine della lista e inserendo l'elemento cambiando i puntatori. L'estrazione e il peek sono implementati prendendo e, nel caso cancellando, il primo valore della lista.

### 2.4 Prestazioni attese

Nelle liste le operazioni che richiedono una ricerca in lunghezza richiedono un tempo proporzionale alla lunghezza della lista, quindi, data lunghezza  $n$ , possiamo aspettarci che nelle liste non ordinate l'estrazione e il peek richiedano  $\Theta(n)$  e nelle liste ordinate l'inserimento richieda  $O(n)$ , in quanto potrebbe finire prima della lunghezza della lista. L'inserimento nelle liste non ordinate e l'estrazione e il peek nelle liste ordinate invece richiede  $O(1)$ , dato che l'elemento è subito disponibile.

Nel max heap invece i tempi sono determinati dalle funzioni che mantengono le proprietà di heap e che scorrono l'heap solo per l'altezza dell'albero ( $\log(n)$ ). Quindi l'inserimento richiederà  $O(\log(n))$  poiché può terminare prima, e l'estrazione  $\Theta(\log(n))$ . Il peek nell'heap richiederà  $O(1)$ .

	Estrazione	Inserimento	Peek
Heap	$\Theta(\log(n))$	$O(\log(n))$	$O(1)$
Lista Ordinata	$O(1)$	$O(n)$	$O(1)$
Lista Non Ordinata	$\Theta(n)$	$O(1)$	$\Theta(n)$

Figura 1: Complessità attese

### 3 Documentazione del codice

#### 3.1 Schema delle classi e scelte implementative

Per prima cosa sono state create le strutture dati atte da sottoporre a test:

- Ogni elemento delle code è stato rappresentato da un PriorityQueueItem con una priorità e un contenuto
- Una interfaccia chiamata PriorityQueueInterface è stata creata da cui derivare le implementazioni
- L'heap è stato implementato con una lista di PriorityQueueItem e le funzioni che servono per mantenerne le proprietà di heap
- Un nodo (Node) racchiude un PriorityQueueItem aggiungendo un puntatore al nodo successivo per poterli usare nelle liste collegate con puntatori
- Una Lista non ordinata è stata implementata con un puntatore a nodo head
- Una Lista ordinata è stata derivata dalla lista non ordinata sovrascrivendo le funzioni per la gestione dei nodi

Sono poi stati scritti due moduli che racchiudono una serie di funzioni per testare e analizzare le implementazioni e per creare i relativi grafici:

- *performance.py* racchiude le funzioni che servono a misurare il tempo impiegato dalle varie implementazioni con le operazioni fondamentali con varie quantità di dati e con vari pattern di priorità.
- *asymptotic\_analysis.py* racchiude le funzioni che servono a creare un grafico che metta in evidenza l'andamento asintotico delle operazioni fondamentali delle varie implementazioni

Le funzioni dei due moduli vengono fatte partire da una funzione main in un file main.py separato.

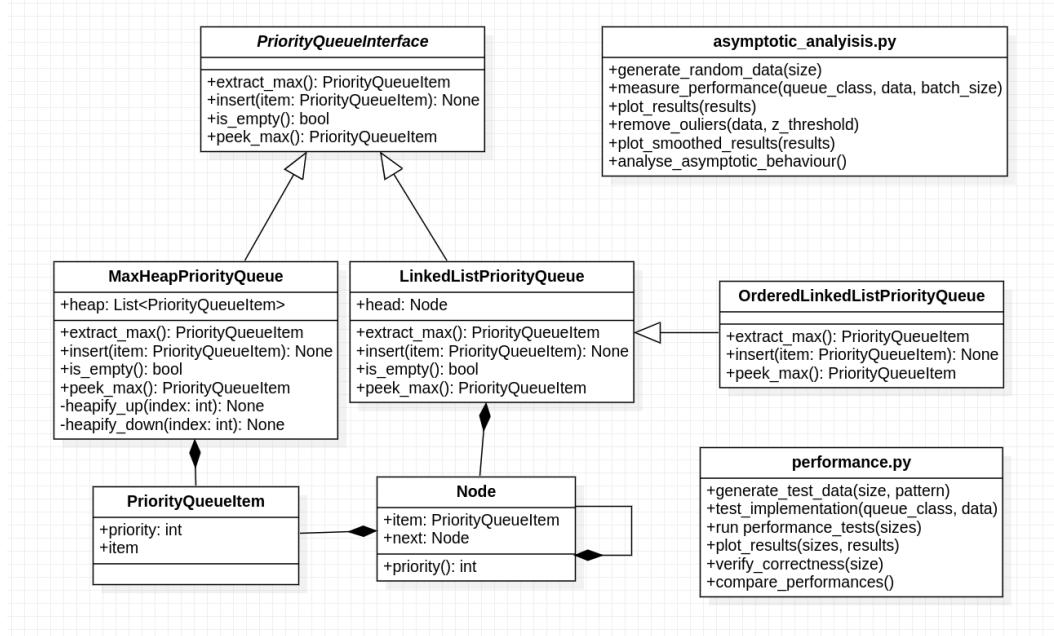


Figura 2: Diagramma delle classi e dei moduli

### 3.2 Descrizione dei metodi principali di ogni classe o modulo

Segue la descrizione dei metodi principali per ogni classe o modulo, vengono tralasciati metodi e funzioni molto semplici come getters e setters.

- **MaxHeapPriorityQueue**

- **insert(), extract\_max() e peek()**

vedi paragrafo 2.3

- **is\_empty() : bool**

Restituisce **True** se la coda è vuota, altrimenti **False**.

- **\_heapify\_up(index: int) : None**

Metodo interno che ripristina la proprietà di heap massimo spostando iterativamente l'elemento con indice **index** verso l'alto. L'elemento continua a essere scambiato con il suo genitore finché la sua priorità è maggiore di quella del genitore o finché raggiunge la radice.

- **\_heapify\_down(index: int) : None**

Metodo interno che ripristina la proprietà di heap massimo spostando iterativamente l'elemento con indice **index** verso il basso. L'elemento viene scambiato con il figlio con priorità maggiore (se presente e con priorità superiore) finché non rispetta la proprietà di heap massimo o raggiunge una posizione foglia.

- **LinkedListPriorityQueue**

- **insert(), extract\_max(), peek()**

vedi paragrafo 2.3

- **is\_empty() : bool**

Restituisce **True** se la coda è vuota, altrimenti **False**.

- **OrderedLinkedListPriorityQueue**

- **insert(), extract\_max(), peek()**

vedi paragrafo 2.3

- **performance.py**

- **compare\_performances()**

Esegue sia il test di correttezza che i test di prestazioni per ogni implementazione. Dopo la verifica della correttezza, esegue i test su dimensioni di input crescenti e genera visualizzazioni dei risultati.

- **verify\_correctness(size: int)**

Verifica la correttezza di ogni implementazione della coda con priorità. Inserisce un set di dati casuale di dimensione **size** in ciascuna coda e verifica che tutti gli elementi siano estratti in ordine decrescente di priorità.

- **generate\_test\_data(size: int, pattern: str) : List[PriorityQueueItem]**

Genera dati di test con dimensione **size** e un **pattern** specificato. I pattern possibili sono: **"random"** per priorità casuali, **"ascending"** per priorità in ordine crescente, e **"descending"** per priorità in ordine decrescente. Ritorna una lista di oggetti **PriorityQueueItem**.

- **run\_performance\_tests(sizes: List[int]) : Dict[str, Dict[str, Dict[str, List[float]]]]**

Esegue test di prestazioni per diverse implementazioni di code con priorità su dimensioni di input crescenti e per ciascuno dei tre pattern di dati. Ritorna un dizionario contenente i risultati dei test.

- **test\_implementation(queue\_class, data: List[PriorityQueueItem]) : tuple[float, float, float]**

Esegue test di prestazioni su una specifica implementazione di coda con priorità. Misura i tempi di esecuzione per le operazioni di inserimento, visualizzazione del massimo, estrazione del massimo e un mix di inserzioni ed estrazioni, restituendo i tempi in una tupla.

- **plot\_results(sizes: List[int], results: Dict[str, Dict[str, Dict[str, List[float]]]])**  
Crea visualizzazioni grafiche dei risultati delle prestazioni. Genera grafici in scala normale e logaritmica per confrontare le prestazioni delle implementazioni delle code con priorità. I grafici sono salvati nelle directory **performance\_normal** e **performance\_log**.
- **asymptotic\_analysis.py**
  - **analyse\_asymptotic\_behaviour()**  
Esegue test di prestazione su implementazioni di coda con priorità con grandi dimensioni di input. Calcola e visualizza i risultati dettagliati e smussati, con una linea di tendenza per ciascuna operazione.
  - **generate\_random\_data(size: int) : List[PriorityQueueItem]**  
Genera una lista di dati casuali di dimensione **size**, ciascuno con una priorità casuale tra 1 e 100000000.
  - **measure\_performance(queue\_class, data: List[PriorityQueueItem], batch\_size: int) : Dict[str, List[float]]**  
Esegue misurazioni di prestazioni per un'implementazione specifica di coda con priorità, inserendo i dati in blocchi di dimensione **batch\_size**. Misura i tempi per le operazioni di inserimento, visualizzazione del massimo e rimozione del massimo, restituendo i tempi in un dizionario.
  - **plot\_results(results: Dict[str, Dict[str, List[float]]])**  
Crea grafici dettagliati per ogni implementazione e ogni operazione. Ogni grafico mostra i tempi di esecuzione con una linea di tendenza. I grafici sono salvati nella directory **detailed\_plots**.
  - **remove\_outliers(data: List[float], z\_threshold: float = 4.0) : List[float]**  
Rimuove i valori anomali da una lista di dati utilizzando il metodo dello **z-score**. I valori con uno **z-score** maggiore della soglia **z\_threshold** vengono esclusi.
  - **plot\_smoothed\_results(results: Dict[str, Dict[str, List[float]]])**  
Crea grafici smussati per ogni implementazione e operazione. Rimuove i valori anomali e applica una media esponenziale per smussare i dati. Aggiunge una linea di tendenza e regola la scala dell'asse y per evidenziare meglio i trend. I grafici sono salvati nella directory **smoothed\_plots**.

### 3.3 Note sull'implementazione

Durante le misurazioni dei tempi disabilitiamo la garbage collection di Python perché si attivava casualmente e andava a sporcare i dati. Vengono applicate alcune accortezze per rendere meglio i risultati nei grafici come andare a impostare la scala sull'asse y in modo da rendere meglio l'andamento se la pendenza della linea di tendenza è molto piccola. Inoltre viene sempre lasciata visibile la linea di tendenza anche quando l'andamento non lo aspettiamo lineare per dare una maggiore indicazione sulla pendenza.

## 4 Descrizione e analisi del primo esperimento - Comparazione di performance

Come già detto nel paragrafo precedente andiamo a misurare le performance delle nostre implementazioni e compararle tra loro

### 4.1 Dati utilizzati

Per ricreare le diverse condizioni in cui vogliamo testare le implementazioni creiamo i dati o in modo randomico, o ascendente o discendente. Per osservare il comportamento all'aumentare delle entrate usiamo le seguenti dimensioni: 50, 100, 250, 500, 750, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000 e 10000. Le implementazioni vengono ogni volta testate con lo stesso set di dati.

### 4.2 Misurazioni

Per prendere le misure di tempo usiamo il metodo `perf_counter()` della libreria `time` e andiamo a misurare il tempo prima e dopo aver svolto l'operazione di interesse. Viene prima misurato l'inserimento di tutti i dati, poi una visualizzazione del minimo tante volte quanto un decimo dei dati, l'estrazione di tutti i dati e infine vengono misurate insieme le operazioni di inserimento e estrazione (in un ordine randomico di 60% inserimenti e 40% estrazioni).

I tempi registrati vengono poi salvati in un dizionario e poi inseriti nei grafici, con scala normale e logaritmica.

In blu sono i risultati del Max Heap, in arancione la lista non ordinata e in verde la lista ordinata.

### 4.3 Risultati sperimentali

#### 4.3.1 valori casuali

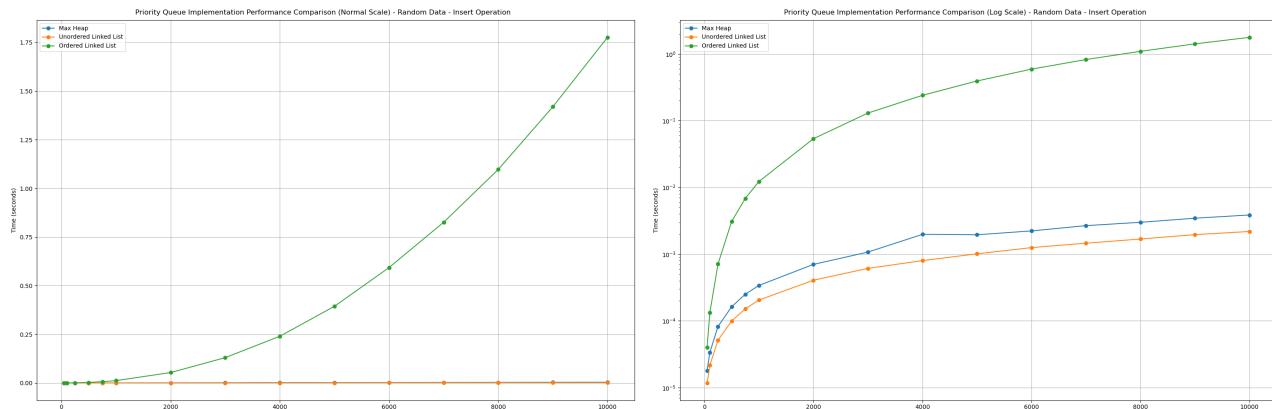


Figura 3: casuali inserimento

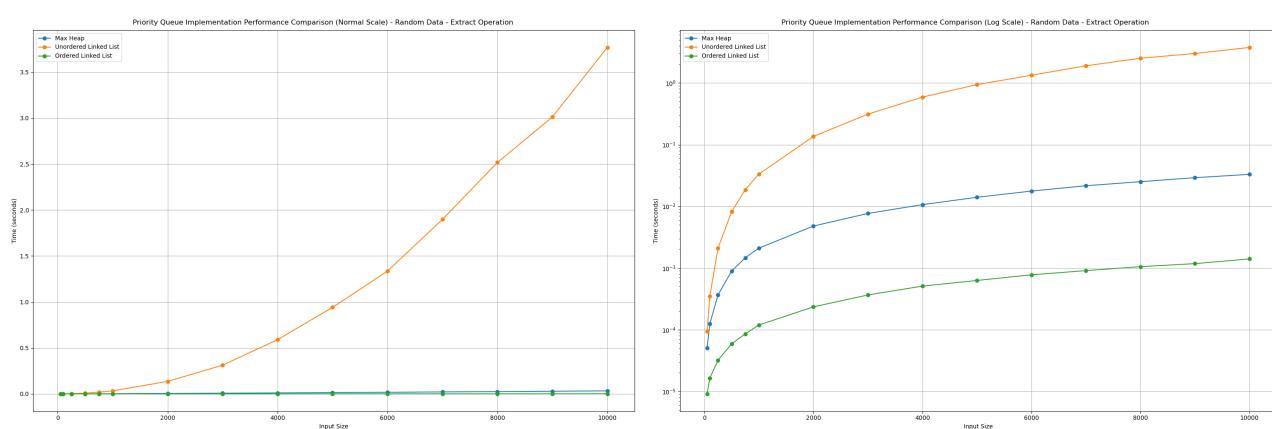


Figura 4: casuali estrazioni

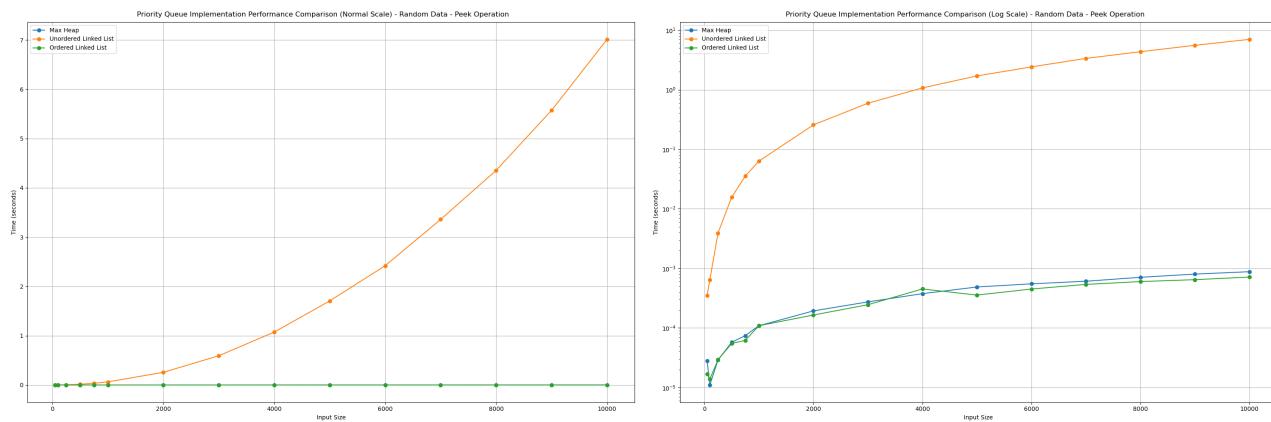


Figura 5: casuali visualizzazione massimo

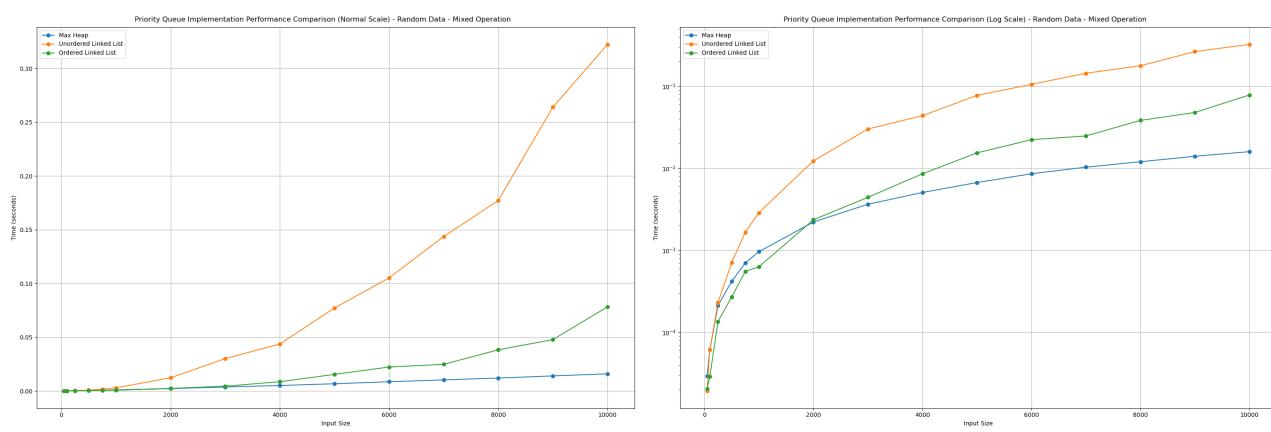


Figura 6: casuali mixto

#### 4.3.2 valori ascendenti

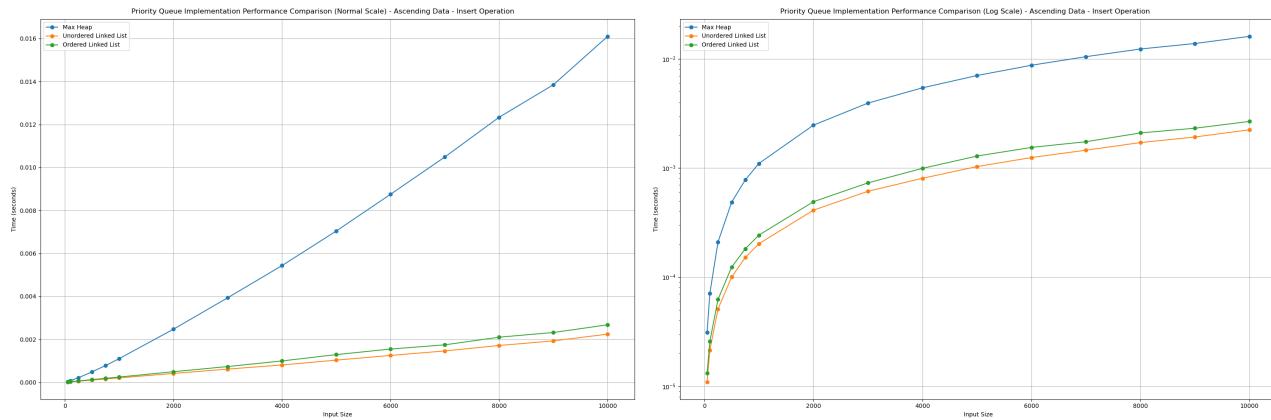


Figura 7: ascendenti inserimento

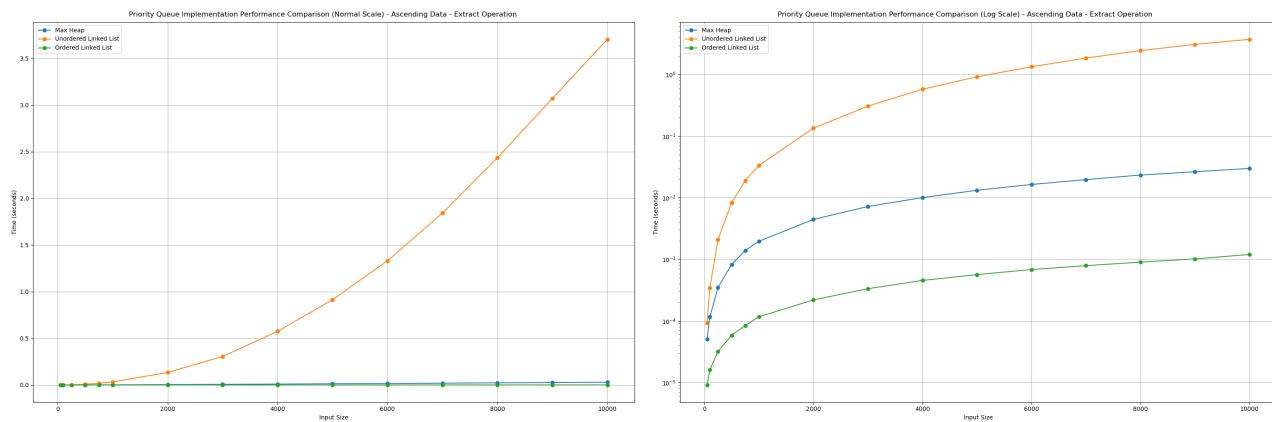


Figura 8: ascendenti estrazione

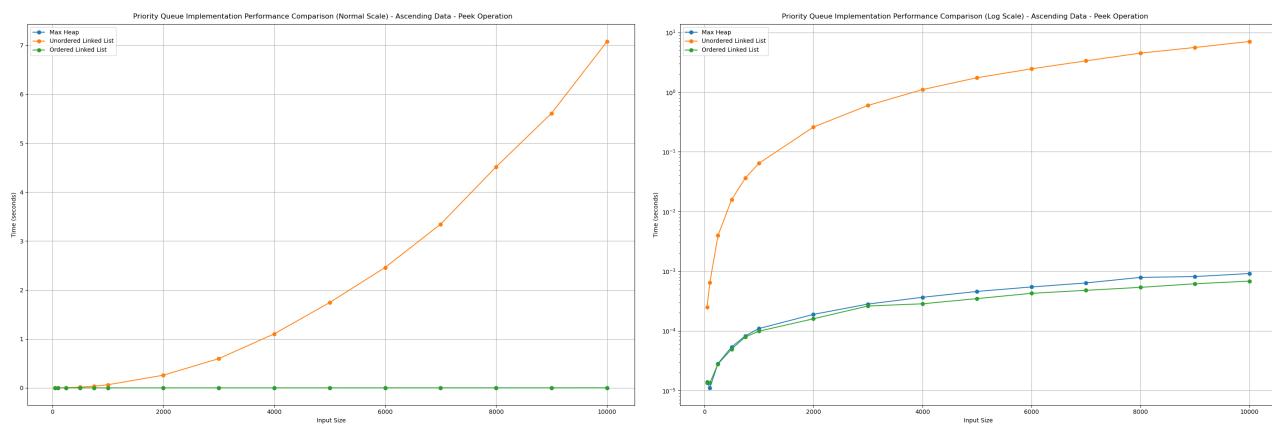


Figura 9: ascendenti visualizzazione massimo

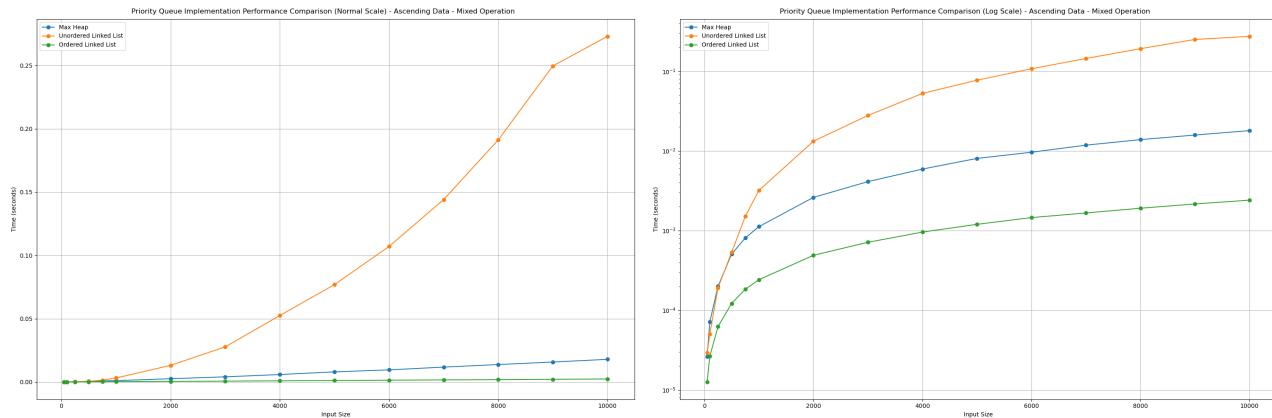


Figura 10: ascendenti misto

### 4.3.3 valori descendenti

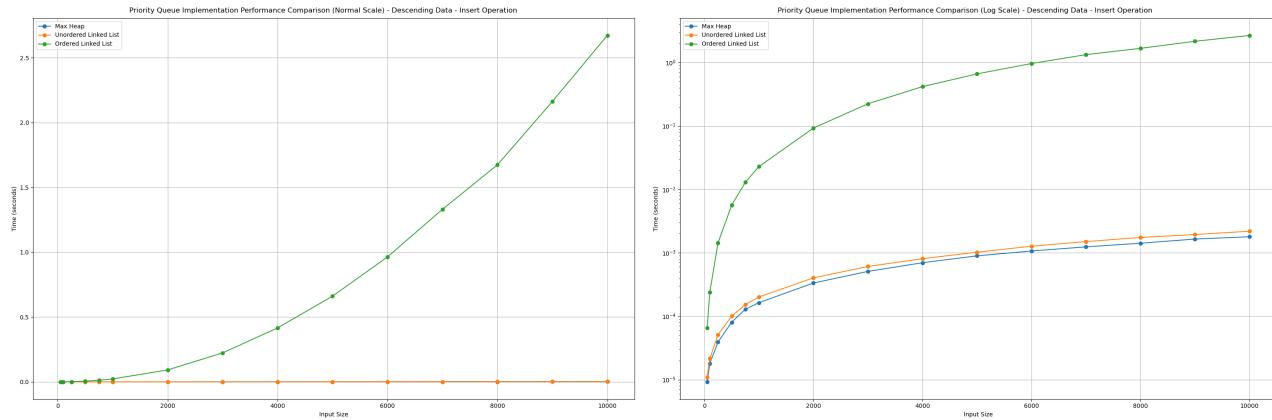


Figura 11: descendenti inserimento

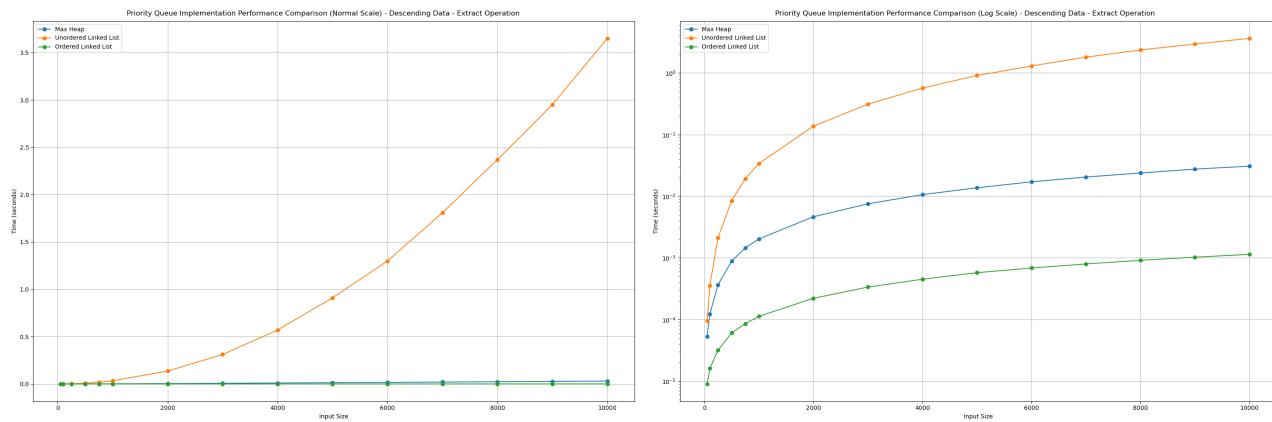


Figura 12: descendenti estrazione

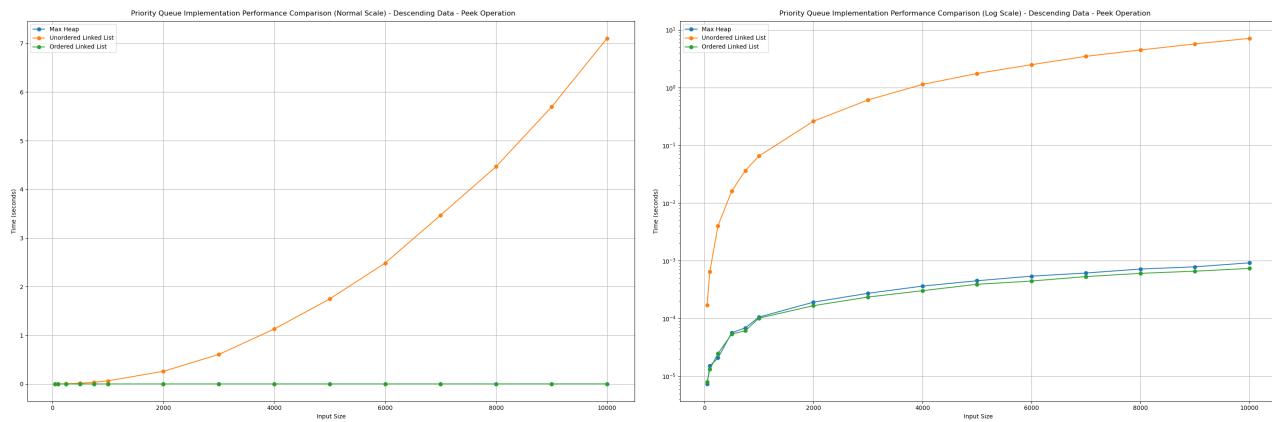


Figura 13: descendenti visualizzazione massimo

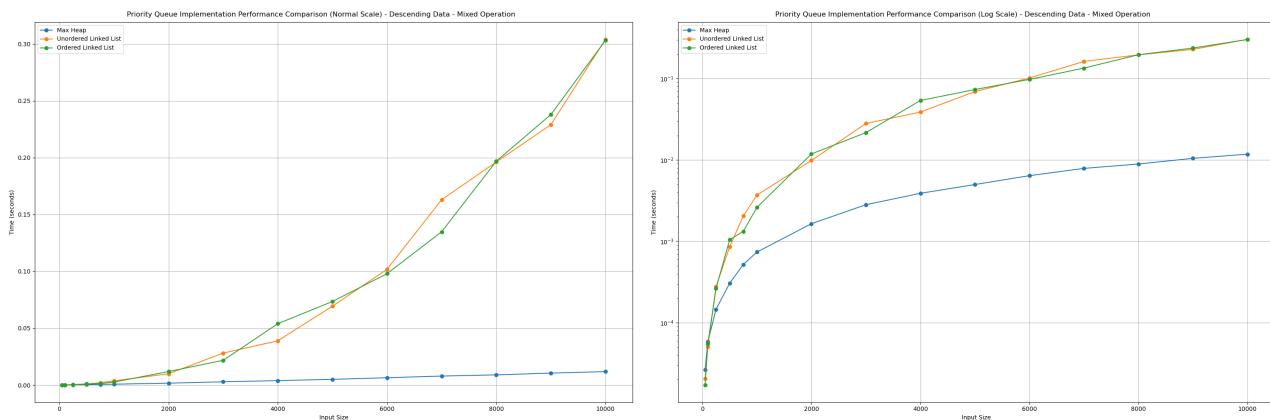


Figura 14: discendenti misto

#### 4.4 Analisi risultati

Nelle condizioni di dati casuali notiamo che al crescere delle dimensioni della coda: la lista ordinata ha un'estrazione più veloce e la lista non ordinata l'inserimento, tuttavia l'heap riesce ad avere performance migliori nelle operazioni miste non avendo il caso peggiore in nessuna delle operazioni, nella visualizzazione del massimo la lista ordinata e l'heap hanno le stesse prestazioni.

Nel caso particolare di dati ordinati in modo crescente la lista ordinata si trova nella sua condizione migliore per l'inserimento, migliorando quindi anche le sue prestazioni miste che dipendevano dalla lentezza dell'inserimento, e quindi battendo l'heap nelle prestazioni.

Nel caso particolare di dati ordinati in modo decrescente la lista ordinata si trova nel caso peggiore per l'inserimento, avendo tempi ben più lunghi di quelle delle altre due implementazioni, i tempi dell'estrazione rimangono gli stessi e quindi l'heap continua ad avere i tempi migliori nel caso misto.

## 5 Descrizione e analisi del secondo esperimento - Analisi asintotica

Andiamo ora ad analizzare le tre implementazioni elemento per elemento cercando di visualizzare l'andamento asintotico al quale tendono.

### 5.1 Dati utilizzati

Per analizzare l'andamento asintotico creiamo dei dati casuali e li analizziamo in lotti così da avere 1000 misurazioni per il grafico, le misurazioni in lotti sono un primo tentativo di diminuire le piccole fluttuazioni casuali di prestazioni e lavorare con tempi più grandi. Siccome non stiamo comparando le varie implementazioni possiamo permetterci di avere dimensioni dei dati diverse per ognuna di esse, valori più significativi sembrano apparire con queste dimensioni di dati:

Max Heap, dimensione totale: 100000, dimensioni lotti: 100,

Lista non ordinata, dimensioni totali: 30000, dimensioni lotti: 30,

Lista ordinata, dimensioni totali: 40000, dimensioni lotti: 40

### 5.2 Misurazioni

Per prendere le misure di tempo usiamo il metodo `perf_counter()` della libreria `time` e andiamo a misurare il tempo prima e dopo aver svolto l'operazione di interesse. Per ogni lotto quindi viene misurato il tempo di inserimento e il tempo per fare tante visualizzazioni del massimo quante le dimensioni del lotto, successivamente, finiti tutti gli inserimenti, per ogni lotto vengono misurate le estrazioni. I dati vengono salvati in un dizionario e inseriti in un grafico.

Si genera poi un secondo insieme di grafici delle prestazioni, con l'obiettivo di evidenziare più chiaramente i trend generali eliminando distorsioni e variazioni anomale. Si eseguono i seguenti passaggi:

- **Rimozione dei valori anomali:** la funzione utilizza il metodo dello *z-score* della libreria `scipy.stats` per identificare e rimuovere i valori anomali che potrebbero influenzare il grafico in modo sproporzionato. Vengono esclusi i punti con una deviazione superiore a una soglia 4.0, garantendo che i dati rimanenti rappresentino più fedelmente il comportamento medio.
- **Media mobile esponenziale:** sui dati senza valori anomali viene applicata una media mobile esponenziale per generare una versione smussata delle tempistiche di esecuzione. Questo approccio assegna un peso decrescente ai dati più vecchi, mantenendo  $\alpha = 0.1$  come fattore di smorzamento. In questo modo, si riducono le oscillazioni e si ottiene una curva che riflette meglio il trend generale senza risentire eccessivamente delle fluttuazioni puntuali. L'algoritmo si basa sulla seguente formula di aggiornamento:

$$\text{nuovo valore} = \alpha \cdot \text{valore\_corrente} + (1 - \alpha) \cdot \text{valore precedente}$$

- **Adattamento dell'asse y:** per migliorare la leggibilità, la funzione regola i limiti dell'asse y, ignorando i primi 50 punti, per concentrarsi sul trend predominante e limitare l'influenza di eventuali variazioni iniziali.

In entrambi i grafici si calcola e sovrappone una linea di tendenza lineare (retta di regressione) sui dati smussati per evidenziare la direzione generale del trend. La pendenza della linea di tendenza è indicata in legenda e in descrizione alle immagini per comprendere la variazione media nel tempo.

Infine, i grafici vengono salvati rispettivamente nelle cartelle `detailed_plots` e `smoothed_plots`.

## 5.3 Risultati sperimentali

### 5.3.1 Max Heap

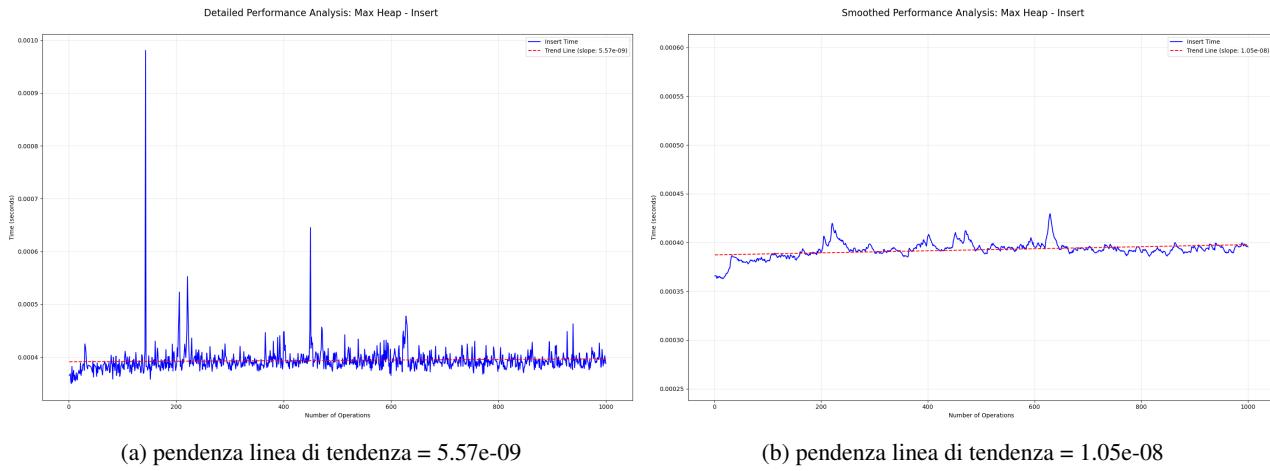


Figura 15: Max Heap Inserimento

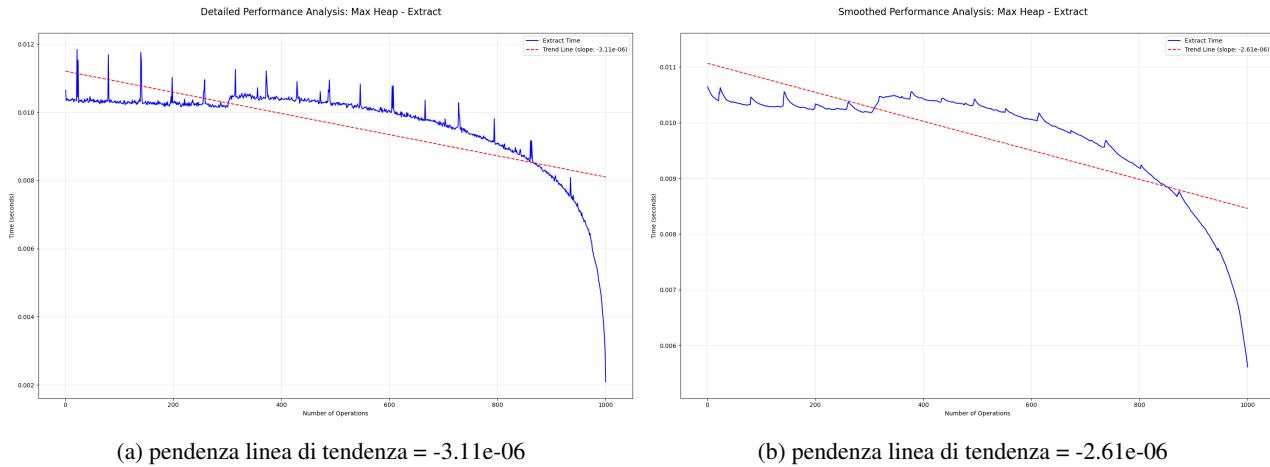


Figura 16: Max Heap estrazione

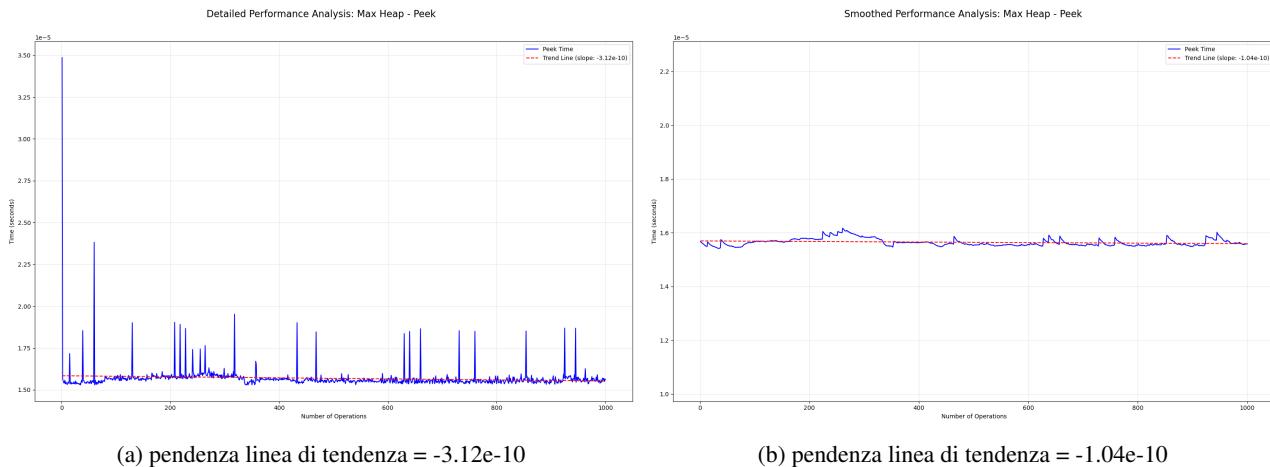


Figura 17: Max Heap visualizzazione massimo

### 5.3.2 Lista non ordinata

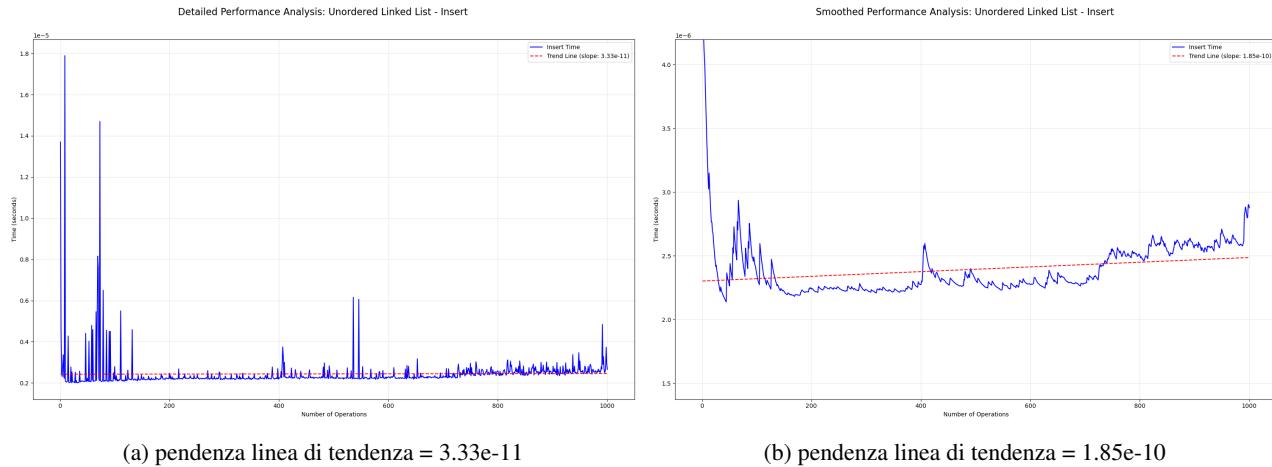


Figura 18: Lista non ordinata Inserimento

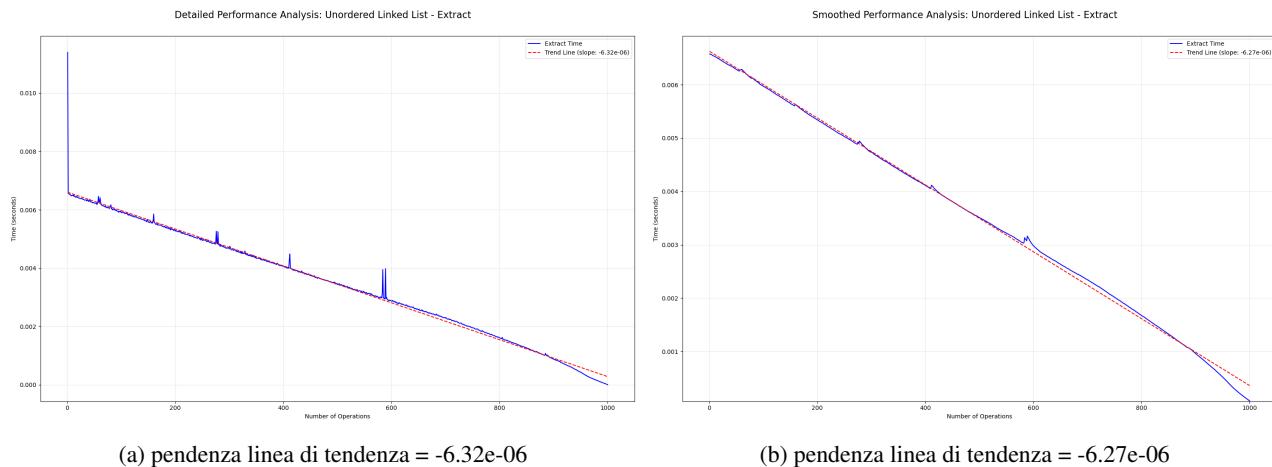


Figura 19: Lista non ordinata estrazione

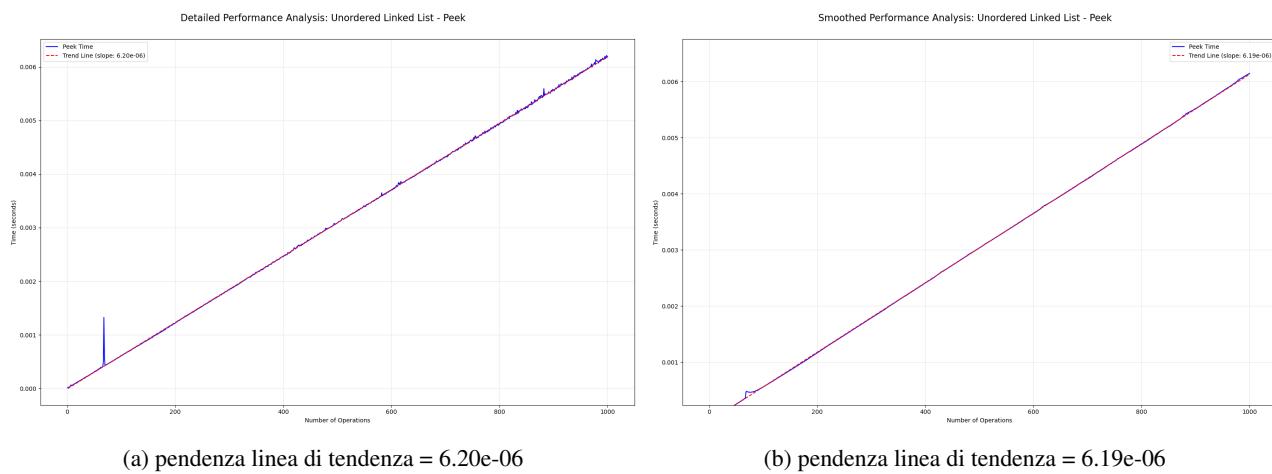


Figura 20: Lista non ordinata visualizzazione massimo

### 5.3.3 Lista ordinata

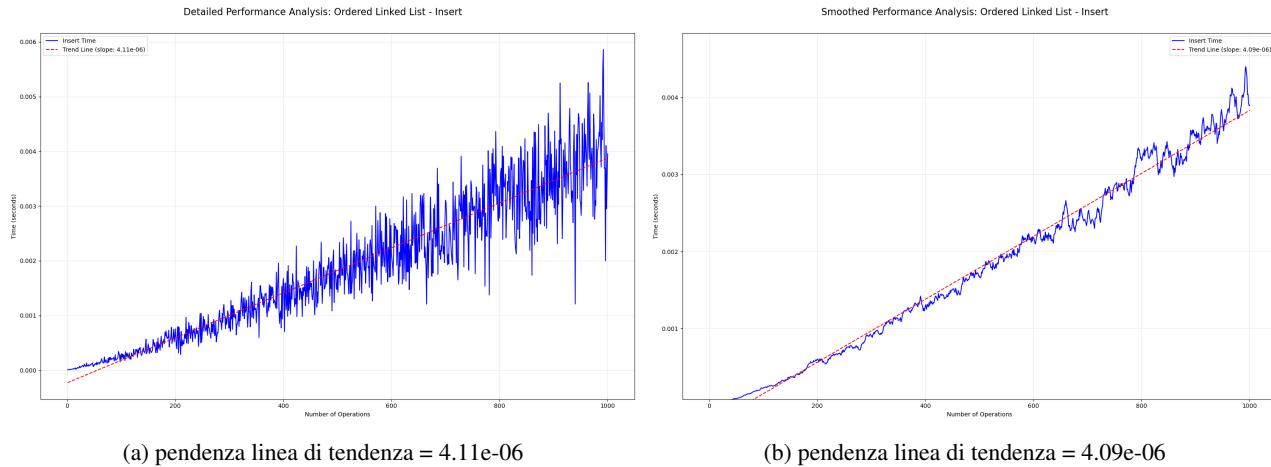


Figura 21: Lista ordinata Inserimento

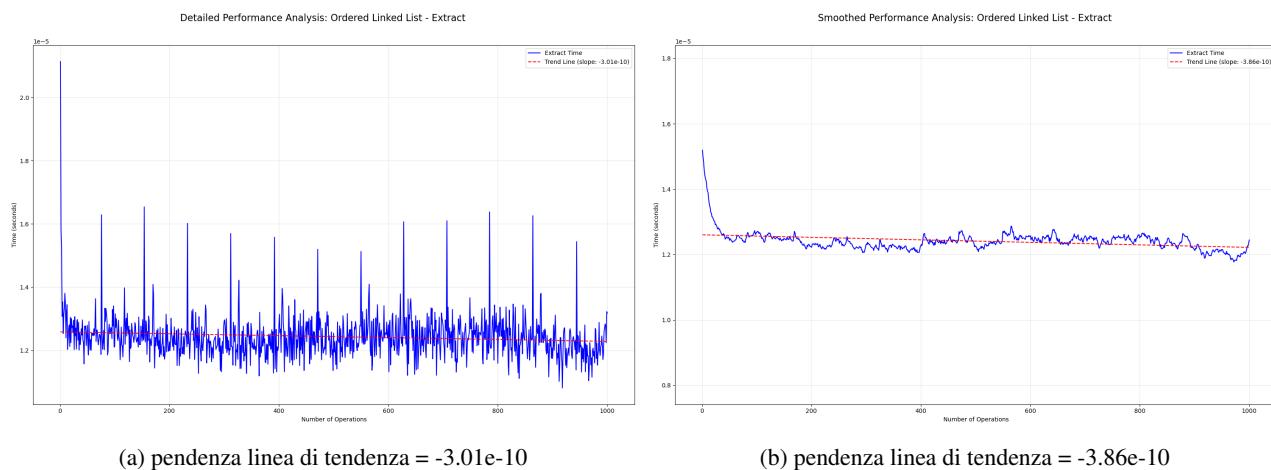


Figura 22: Lista ordinata estrazione

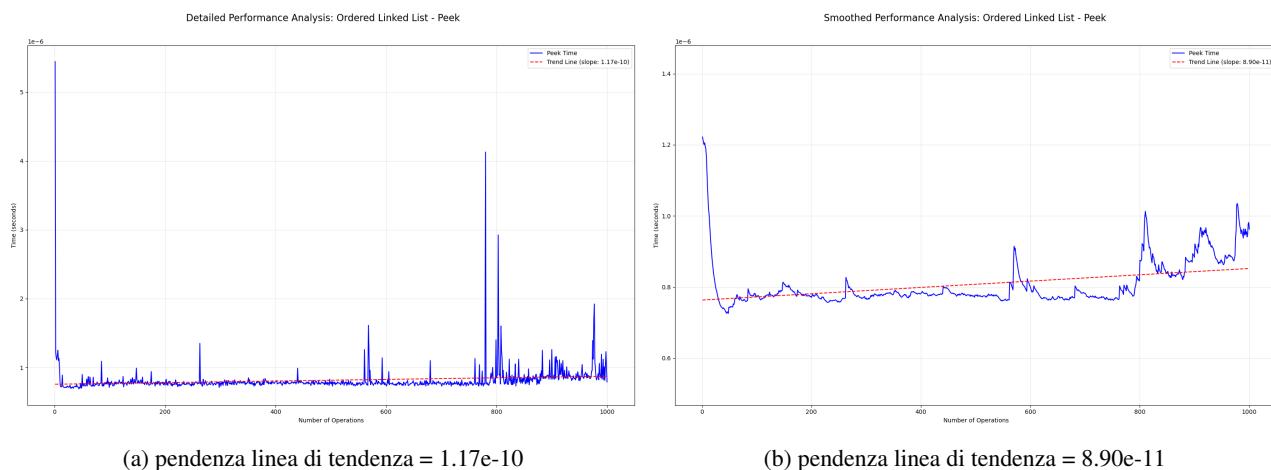


Figura 23: Lista ordinata visualizzazione massimo

## 5.4 Analisi risultati

I grafici sembrano rispettare le predizioni 1 che abbiamo fatto in merito agli andamenti asintotici, talvolta più visibili nei grafici originali talvolta in quelli smussati.

L'unica particolarità è l'operazione di inserimento del Max heap che, sebbene rispetti la complessità  $O(\log(n))$  non arriva a comportarsi come il suo caso peggiore  $\log(n)$  ma ha un comportamento molto più vicino a quello costante e quindi migliore di quello atteso.

## 6 Conclusioni

I risultati sperimentali hanno confermato gran parte delle previsioni teoriche sulle prestazioni delle diverse implementazioni di code di priorità, con alcune osservazioni notevoli:

- Il Max Heap si è dimostrato l'implementazione più bilanciata, non presentando casi particolarmente negativi in nessuna operazione e risultando particolarmente efficace in scenari di utilizzo misto.
- La Lista Ordinata, eccellente nelle operazioni di estrazione e visualizzazione del massimo ma inefficiente nell'inserimento, ha mostrato prestazioni competitive con il Max Heap solo con dati in ordine crescente.
- La Lista Non Ordinata, nonostante gli inserimenti veloci, ha confermato le sue limitazioni nelle operazioni di ricerca del massimo.

L'analisi asintotica ha rivelato un interessante comportamento dell'inserimento nel Max Heap, che nella pratica si è dimostrato più vicino al caso costante rispetto al teorico  $O(\log n)$ . Questo suggerisce che l'implementazione è particolarmente efficiente per questa operazione. Le altre implementazioni si sono comportate come da predizioni ovvero:

	Estrazione	Inserimento	Peek
Heap	$\Theta(\log(n))$	$O(\log(n))$	$O(1)$
Lista Ordinata	$O(1)$	$O(n)$	$O(1)$
Lista Non Ordinata	$\Theta(n)$	$O(1)$	$\Theta(n)$

In conclusione, il Max Heap si conferma come la scelta più versatile per la maggior parte delle applicazioni, mentre le implementazioni basate su liste possono risultare vantaggiose solo in scenari molto specifici dove predomina un particolare tipo di operazione.