UNIVERSITÀ
DEGLI STUDI
FIRENZE

UNIVERSITÀ DEGLI STUDI DI FIRENZE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Analisi Comparativa di Diverse Implementazioni di Code di Priorità

Autore:
Ciampalini Dania

N° Matricola:
7133412

Corso principale:
Algoritmi e Strutture Dati

Docente corso:
Simone Marinai

Contents

List of Figures

1 Introduzione generale

1.1 Breve descrizione dello svolgimento degli esercizi

Per ogni esercizio suddividiamo la sua descrizione in 4 parti fondamentali:

- **Spiegazione teorica del problema**: in questa sezione viene fornita una descrizione formale del problema da risolvere, basata sugli assunti teorici affrontati nel corso e presenti nel libro di testo di Algoritmi e Strutture Dati [1].
- **Documentazione del codice**: qui si illustra nel dettaglio l'implementazione dell'esercizio, evidenziando le scelte progettuali, la struttura del codice e le eventuali ottimizzazioni adottate.
- **Descrizione degli esperimenti condotti**: a partire dal codice sviluppato, si eseguono una serie di misurazioni volte a verificare sperimentalmente le ipotesi teoriche formulate in precedenza.
- **Analisi dei risultati sperimentali**: si analizzano criticamente i dati raccolti durante la fase sperimentale, confrontandoli con le previsioni teoriche, infine si formulano osservazioni e considerazioni conclusive.

1.2 Specifiche della piattaforma di test

La piattaforma di test sarà la stessa per ogni esercizio che vedremo. L'hardware del computer usato per testare questi esercizi è il seguente:

- **CPU**: 1,8 GHz Intel Core i5 dual-core
- **RAM**: 8 GB 1600 MHz DDR3
- **SSD**: APPLE SSD SM0128G Media 121GB

Il linguaggio di programmazione utilizzato sarà Python e la piattaforma in cui il codice è stato scritto e testato è l'IDE **PyCharm 2025.1.1.1**. La stesura di questo testo è avvenuta tramite l'utilizzo dell'editor online **Overleaf**.

Part I
Implementazioni di Code di Priorità

Esercizio 1

- Vogliamo analizzare le differenze tra diverse implementazioni di code di priorità:
  - Heap
  - Lista concatenata
  - Lista concatenata ordinata
- **Nota**: La lista deve essere implementata considerando strutture collegate con puntatori e non la struttura dati lista di Python.

2 Spiegazione teorica del problema

2.1 Introduzione

In questa sezione si analizzano diverse implementazioni delle code di priorità, con l'obiettivo di metterle a confronto attraverso lo studio della loro complessità computazionale. L'attenzione è focalizzata su tre operazioni fondamentali — **inserimento**, **ricerca** ed **estrazione del valore con priorità massima** — in quanto costituiscono le principali funzionalità offerte da una coda di priorità e sono le più frequentemente utilizzate in applicazioni reali. Le strutture considerate includono l'**heap binario**, la **lista concatenata** e la **lista concatenata ordinata**, con la precisazione che per le implementazioni basate su liste si farà uso esclusivo di strutture dinamiche collegate mediante puntatori, escludendo l'impiego della struttura dati list offerta nativamente dal linguaggio Python.

2.2 Definizione struttura dati

Una **coda di priorità** è una struttura dati che serve a mantenere un insieme dinamico S di elementi, ciascuno con un valore associato detto **chiave**. Ci sono due tipi di code di priorità: code di **max-priorità** e code di **min-priorità**. Una coda di max-priorità supporta le seguenti operazioni:

- `INSERT(S, x, k)` inserisce l'elemento x con chiave k nell'insieme S;
- `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 grande almeno quanto il valore attuale della chiave dell'elemento x.

Le code di min-priorità supportano operazioni analoghe, con la differenza che gli elementi vengono gestiti in ordine crescente rispetto ai valori di priorità.

Poiché le differenze implementative sono minime e simmetriche rispetto a quelle delle code di max-priorità, in questo esperimento ci concentreremo esclusivamente sull'analisi delle prestazioni delle code di *max-priorità*.

2.3 Criteri di selezione delle operazioni sperimentali

Nel contesto del nostro esperimento, ci concentreremo sulle tre operazioni fondamentali: `INSERT`, `MAXIMUM` e `EXTRACT-MAX`. Queste sono sufficienti a mettere in luce le differenze strutturali e di prestazioni tra le varie implementazioni delle code di priorità.

L'operazione `INCREASE-KEY` risulta meno adatta in un confronto generico tra strutture dati, in quanto la sua implementazione in strutture come la lista concatenata diventa inefficiente o poco chiara senza una struttura di supporto (come un dizionario di puntatori), introducendo un disallineamento rispetto alle altre operazioni in termini di semplicità e coerenza del confronto.

In scenari pratici, `INCREASE-KEY` è particolarmente rilevante solo in contesti specifici (es. Dijkstra), dove le strutture di priorità devono gestire aggiornamenti efficienti delle distanze stimate.

Tuttavia, tali casi richiedono anche un meccanismo di tracciamento degli elementi all'interno della coda che esula dallo scopo di questo confronto sperimentale.

Per questi motivi, si è deciso di non includere `INCREASE-KEY` tra le operazioni oggetto dell'analisi empirica.

2.4 Nomenclatura e convenzioni

Nel corso dell'esperimento adotteremo alcune convenzioni terminologiche e simboliche utili alla chiarezza e alla formalizzazione dei risultati:

Notazione asintotica Per descrivere la complessità computazionale delle operazioni si usano le notazioni classiche della teoria degli algoritmi:

- $\mathcal{O}(f(n))$: limite superiore asintotico (complessità nel caso peggiore).
- $\Theta(f(n))$: ordine di crescita esatto (quando il caso peggiore e il caso migliore coincidono asintoticamente).

Simboli ricorrenti Nel testo useremo:

- n: il numero totale di elementi contenuti nella struttura dati al momento considerato.
- h: l'altezza della struttura (nel caso dell'heap, $h = \lfloor \log_2 n \rfloor$).

2.5 Panoramica delle strutture analizzate

Nel presente esperimento si analizzano e confrontano tre diverse implementazioni di code di priorità:

- **Heap binario**
- **Lista concatenata non ordinata (LL)**
- **Lista concatenata ordinata (SLL)**

Heap binario

L'heap binario è una struttura dati che rappresenta un albero binario quasi completo, implementato tramite array. Le sue principali proprietà sono:

- Ogni nodo ha al massimo due figli.
- L'albero è completo fino all'ultimo livello, riempito da sinistra verso destra.
- Essendo un albero binario, l'heap è dotato di un'altezza pari a: $h = \lfloor \log_2 n \rfloor$.
- In un *max-heap*, ogni nodo ha una chiave maggiore o uguale a quelle dei suoi figli.

Prestazioni teoriche:

- `INSERT`: $\mathcal{O}(\log n)$ nel caso medio e peggiore poichè si deve scorrere tutta l'altezza dell'heap, $\mathcal{O}(1)$ nel caso migliore cioè se l'elemento va in fondo.
- `MAXIMUM`: $\mathcal{O}(1)$ in tutti i casi poichè il massimo è sempre alla radice.
- `EXTRACT-MAX`: $\mathcal{O}(\log n)$ nel caso medio e peggiore, $\mathcal{O}(1)$ nel caso migliore se heap ha solo un elemento o la ristrutturazione è minima.

Differenze rispetto alle altre: L'heap offre un buon compromesso fra le varie operazioni, risultando generalmente bilanciato. Rispetto alle liste offre prestazioni più costanti in contesti di utilizzo intensivo.

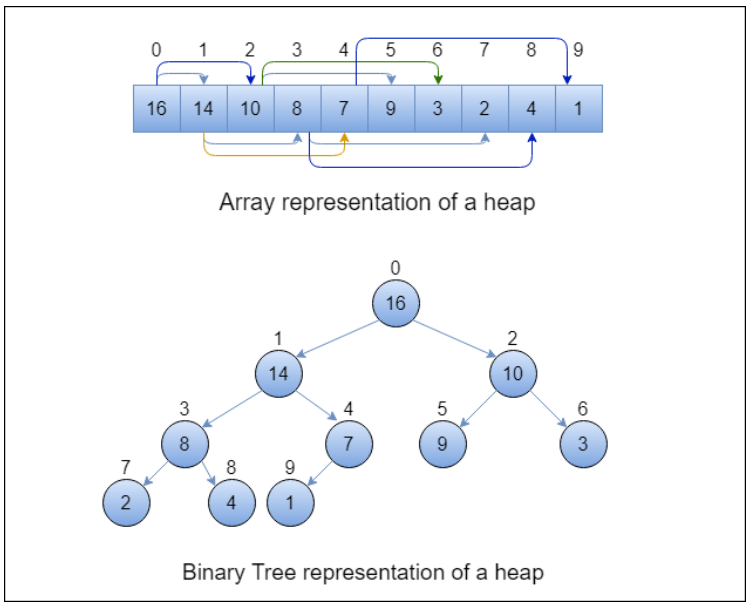

Figure 1: Heap binario con max-heap

Lista concatenata non ordinata (LL o LC)

In questa struttura ogni nuovo elemento viene inserito in testa alla lista, senza alcun ordinamento.

Prestazioni teoriche:

- `INSERT`: $\mathcal{O}(1)$ in tutti i casi in quanto si fa semre l'inserimento in testa.
- `MAXIMUM`: $\mathcal{O}(n)$ nel caso medio e peggiore, ovvero quano si deve scorrere tutta la lista, $\mathcal{O}(1)$ nel caso migliore se il massimo è già in testa.
- `EXTRACT-MAX`: $\mathcal{O}(n)$ nel caso medio e peggiore, $\mathcal{O}(1)$ nel caso migliore (se il massimo è in testa).

Differenze rispetto alle altre: Semplice da implementare, ma inefficiente per accessi e rimozioni. È indicata solo in scenari con molti inserimenti e poche estrazioni.

Lista concatenata ordinata (SLL O LCO)

Questa lista mantiene gli elementi ordinati per priorità decrescente; l'elemento massimo si trova sempre in testa.

Prestazioni teoriche:

- `INSERT`: $\mathcal{O}(n)$ nel caso medio e peggiore cioè se l'inserimento avviene in coda o nel mezzo, $\mathcal{O}(1)$ nel caso migliore (inserimento in testa).
- `MAXIMUM`: $\mathcal{O}(1)$ in tutti i casi perchè è sempre l'elemento in testa.
- `EXTRACT-MAX`: $\mathcal{O}(1)$ in tutti i casi (rimozione in testa).

Differenze rispetto alle altre: È esattamente il contrario della lista non ordinata: costosa per inserimenti ma ottimale per accesso e rimozione del massimo.

2.6 Assunzioni e ipotesi sperimentali

L'obiettivo del nostro esperimento è verificare sperimentalmente la corrispondenza tra le complessità teoriche e i tempi di esecuzione reali osservati. Ci aspettiamo i seguenti comportamenti:

- L'heap dovrebbe garantire buone prestazioni complessive, con costi logaritmici per le operazioni principali, anche nei casi peggiori.
- La lista non ordinata dovrebbe eccellere in `INSERT`, ma degradare fortemente in `EXTRACT-MAX` e `MAXIMUM`, soprattutto in scenari dove il massimo si trova in fondo.

- La lista ordinata dovrebbe penalizzare `INSERT` per garantire `MAXIMUM` e `EXTRACT-MAX` in tempo costante. Tuttavia, la sua efficienza all'inserimento può variare molto a seconda della posizione dove va inserito il nuovo elemento.

Di seguito si riporta una tabella comparativa delle complessità teoriche delle operazioni, nel caso medio, per ciascuna implementazione di coda con priorità:

Struttura	`INSERT`	`MAXIMUM`	`EXTRACT-MAX`
Heap binario	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$
Lista non ordinata	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Lista ordinata	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$

Table 1: Complessità teoriche delle operazioni nel caso medio per le tre strutture analizzate

Il nostro obiettivo in questo esperimento è verificare sperimentalmente la veridicità delle complessità teoriche riportate nella Table 1, e determinare in quali condizioni una certa implementazione di coda con priorità risulti più efficiente rispetto alle altre.

3 Documentazione del codice

Per valutare l'efficienza di diverse implementazioni di code di priorità, abbiamo sviluppato tre classi principali: **MaxHeap**, **LinkedList** e **SortedLinkedList**. Ognuna di queste classi incarna un approccio distinto per la gestione delle operazioni fondamentali di una coda di priorità: inserimento di un elemento, estrazione dell'elemento con priorità massima e consultazione dell'elemento con priorità massima.

3.1 Implementazione tramite Max-Heap

La classe **MaxHeap** realizza una coda di priorità utilizzando un array, conformemente agli algoritmi standard per la gestione di uno heap binario massimizzato, come descritto nella letteratura algoritmica di riferimento. Le operazioni chiave, quali `max_heapify` (ripristino della proprietà di max-heap), `heap_extract_max` (estrazione del massimo), `heap_increase_key` (incremento della chiave) e `max_heap_insert` (inserimento con mantenimento della proprietà di max-heap), sono implementate seguendo fedelmente le procedure algoritmiche consolidate.

3.2 Implementazioni basate su Liste Concatenate

La classe **Node**, definita nel modulo `node.py`, costituisce l'elemento fondamentale per la costruzione delle strutture **LinkedList** e **SortedLinkedList**. La relazione tra le liste e i nodi è di tipo aggregazione: ogni lista contiene un riferimento al primo nodo (`head`), e ciascun nodo contiene un puntatore al nodo successivo (`next`).

3.2.1 Lista Concatenata (Non Ordinata)

La classe **LinkedList** implementa una lista concatenata semplice in cui gli elementi non sono mantenuti in un ordine specifico. Come anticipato, si avvale della classe **Node** per rappresentare i singoli elementi della lista. In questa implementazione, l'inserimento di un nuovo elemento avviene sempre in testa alla lista. Al contrario, l'identificazione e la successiva rimozione dell'elemento massimo richiedono una scansione sequenziale dell'intera lista.

3.2.2 Lista Concatenata Ordinata

La classe **SortedLinkedList** eredita le funzionalità di base da **LinkedList** (*ereditarietà*) ma ne ridefinisce il comportamento dell'operazione di inserimento. L'obiettivo è mantenere gli elementi della lista ordinati in modo decrescente in base alla loro priorità. Questa strategia comporta un'ottimizzazione delle operazioni `maximum` (consultazione del massimo) e `extract_max` (estrazione del massimo), poiché l'elemento con la priorità più alta si trova sempre in testa alla lista (`head`). Tuttavia, il mantenimento dell'ordinamento introduce un costo computazionale maggiore per l'operazione di inserimento, che ora richiede una ricerca della posizione corretta all'interno della lista.

3.3 Visualizzazione e Confronto delle Prestazioni

Per facilitare la visualizzazione e il confronto delle prestazioni delle diverse implementazioni, è stata sviluppata la classe **PlotGenerator**. Questa classe ha il compito di raccogliere e aggregare i tempi medi di esecuzione per ciascuna operazione fondamentale (inserimento, estrazione del massimo) e per diverse dimensioni del set di dati. I risultati di questa analisi vengono quindi presentati graficamente, salvati in formato PDF e visualizzati a schermo per una più agevole interpretazione.

3.4 Diagramma UML delle Classi

Il diagramma UML seguente illustra le relazioni strutturali tra le classi definite nel sistema:

come si può osservare dal diagramma, **SortedLinkedList** è una specializzazione di **LinkedList** (relazione di ereditarietà indicata da una freccia con punta a triangolo vuota). Entrambe le classi **LinkedList** e **SortedLinkedList** utilizzano la classe **Node** attraverso una relazione di aggregazione (indicata da un rombo vuoto lato **LinkedList/SortedLinkedList**).

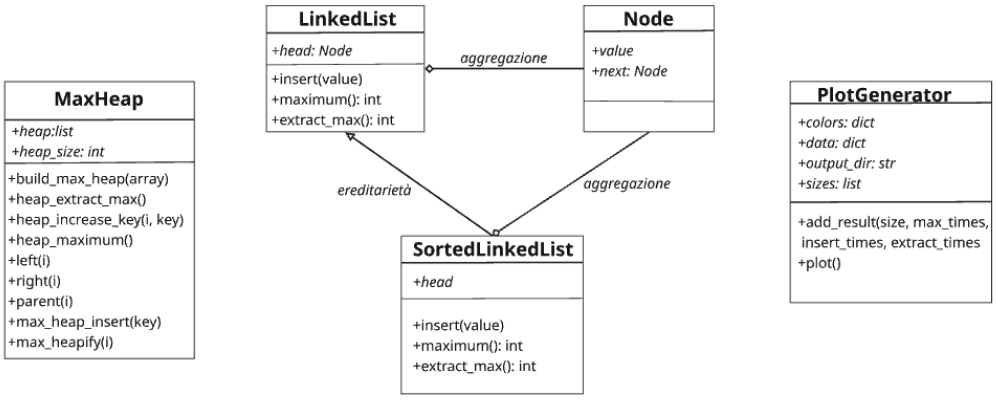

Figure 2: Diagramma UML delle classi

Come si può osservare, la classe `SortedLinkedList` estende `LinkedList`, mentre entrambe le strutture utilizzano `Node` per costruire la lista concatenata. Infine, la classe **PlotGenerator** è responsabile della visualizzazione dei risultati, senza una relazione diretta di composizione o ereditarietà con le strutture dati implementate.

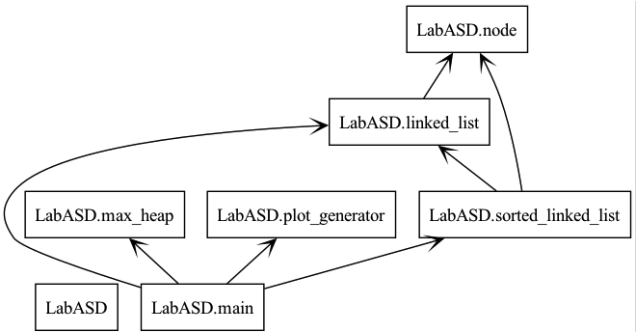

Figure 3: Relazione tra i moduli del progetto[1]

3.5 Descrizione dei metodi implementati

In questa sezione, si procederà con la descrizione dettagliata delle funzionalità di ciascun metodo implementato all'interno delle classi **LinkedList**, **SortedLinkedList** e **MaxHeap**, che costituiscono le diverse strategie per l'implementazione di una coda di priorità analizzate in questo studio.

- **LinkedList**
  - **__init__()**: Inizializza una nuova lista concatenata impostando l'attributo `head` a `None`, indicando una lista inizialmente vuota.
  - **insert(value)**: Crea un nuovo nodo con il valore fornito e lo inserisce all'inizio della lista. Il puntatore `next` del nuovo nodo viene impostato sulla vecchia testa e la `head` della lista viene aggiornata al nuovo nodo.
  - **maximum()**: Restituisce il valore massimo presente nella lista. Se la lista è vuota, solleva un'eccezione. Itera attraverso la lista mantenendo traccia del valore massimo incontrato.
  - **extract_max()**: Estrae e restituisce il valore massimo dalla lista. Se la lista è vuota, solleva un'eccezione. Individua il nodo contenente il valore massimo, aggiorna i puntatori per rimuoverlo dalla lista (gestendo anche il caso in cui il massimo sia in testa) e restituisce il valore estratto.

1

[1] Questo schema è stato realizzato usando la libreria integrata Pylint.

- **SortedLinkedList** (eredita da **LinkedList**)
  - **insert(value)**: Esegue l'inserimento di un nuovo nodo mantenendo la lista ordinata in modo decrescente. Se la lista è vuota o il nuovo valore è maggiore della testa, il nuovo nodo viene inserito in testa. Altrimenti, scorre la lista fino a trovare la posizione corretta per inserire il nuovo nodo, preservando l'ordinamento.
  - **maximum()**: Restituisce il valore massimo presente nella lista ordinata. Poiché la lista è mantenuta in ordine decrescente, il massimo si trova sempre in testa e viene restituito direttamente (solleva un'eccezione se la lista è vuota).
  - **extract_max()**: Estrae e restituisce il valore massimo dalla lista ordinata. Anche in questo caso, il massimo è in testa. La testa viene aggiornata al nodo successivo e il valore massimo viene restituito (solleva un'eccezione se la lista è vuota).
- **MaxHeap**
  - **__init__()**: Inizializza un nuovo max-heap come una lista vuota e imposta la dimensione dell'heap a zero.
  - **left(i)**: Restituisce l'indice del figlio sinistro del nodo all'indice *i*.
  - **right(i)**: Restituisce l'indice del figlio destro del nodo all'indice *i*.
  - **parent(i)**: Restituisce l'indice del padre del nodo all'indice *i*.
  - **max_heapify(i)**: Mantiene la proprietà del max-heap a partire dal nodo all'indice *i*. Confronta il nodo con i suoi figli e, se necessario, lo scambia con il figlio più grande, ripetendo l'operazione sul sottoalbero interessato.
  - **build_max_heap(array)** : Costruisce un max-heap da un array dato. Inizializza l'heap con l'array e poi applica `max_heapify` a partire dall'ultimo nodo non foglia fino alla radice.
  - **heap_maximum()** : Restituisce il valore massimo (radice) del max-heap (solleva un'eccezione se l'heap è vuoto).
  - **heap_extract_max()** : Estrae e restituisce il valore massimo (radice) dal max-heap. Sostituisce la radice con l'ultimo elemento, decrementa la dimensione dell'heap e ripristina la proprietà del max-heap sulla nuova radice (solleva un'eccezione se l'heap è vuoto).
  - **heap_increase_key(i, key)** : Aumenta il valore della chiave del nodo all'indice *i* al nuovo valore *key*, mantenendo la proprietà del max-heap risalendo l'albero e scambiando il nodo con il suo padre se necessario (solleva un'eccezione se la nuova chiave è minore di quella corrente).
  - **max_heap_insert(key)** : Inserisce una nuova chiave nel max-heap. Aumenta la dimensione dell'heap, aggiunge un valore temporaneo ($-\infty$) e poi utilizza `heap_increase_key` per posizionare correttamente la nuova chiave.

3.6 Descrizione dei metodi per la Visualizzazione e l'Analisi delle Prestazioni

Questa sezione illustra le funzionalità dei metodi implementati nella classe **PlotGenerator** e nelle funzioni principali del modulo `main.py` utilizzate per la misurazione e l'analisi delle prestazioni delle strutture dati.

- **Classe `PlotGenerator`**
  - **__init__(output_dir="grafici_output")**: Inizializza un oggetto **PlotGenerator**. Questa operazione crea la directory specificata da `output_dir` se non esiste, dove verranno salvati tutti i grafici generati.
  - **plot_times(sizes, data_dict, title="", xlabel="", ylabel="", filename=None, show=False)**: Genera un grafico a linee che mostra i tempi di esecuzione per diverse strutture dati o diverse operazioni, in funzione della dimensione dell'input (`sizes`). Il parametro `data_dict` è un dizionario le cui chiavi sono le etichette delle curve (es. nomi delle strutture dati o operazioni) e i valori sono liste dei tempi misurati. Permette di personalizzare titolo, etichette degli assi e il nome del file di output.

* **plot_single_curve(sizes, y_values, title="", xlabel="", ylabel="", filename=None, show=False, y_ref=None)**: Genera un grafico a linea singola, utile per visualizzare il rapporto $T(n)/f(n)$ o la complessità costante. Accetta una lista di dimensioni `sizes` e i corrispondenti valori `y_values` da plottare. Include la possibilità di aggiungere una linea di riferimento orizzontale (`y_ref`) per evidenziare un valore medio o una costante asintotica, regolando dinamicamente i limiti dell'asse Y per un miglior focus.

– **Funzioni Principali nel `main.py`**

* **media_mobile(lista, finestra=3)**: Calcola la media mobile di una lista di valori utilizzando una finestra specificata. Viene impiegata per smussare le curve dei tempi di esecuzione nei grafici, rendendoli più leggibili e meno soggetti a fluttuazioni casuali.
* **benchmark_structure(structure_class, sizes, repeats=5)**: Misura i tempi di esecuzione delle operazioni di `Insert`, `Maximum` ed `Extract Max` per una data classe di struttura dati (`structure_class`) su un range di dimensioni dell'input specificato da `sizes`. Ogni misurazione viene ripetuta `repeats` volte per ottenere una media più robusta. Restituisce tre liste contenenti i tempi medi per singola operazione (divisi per n) per inserimento, massimo ed estrazione.
* **calcola_rapporti(dati, sizes, tipo="log")**: Calcola il rapporto tra i tempi di esecuzione misurati (`dati`) e una funzione di complessità asintotica specificata dal parametro `tipo` (es. $\log n$, n, o costante). Questo metodo è fondamentale per verificare sperimentalmente se il comportamento osservato delle operazioni segue la complessità teorica attesa ($T(n)/f(n) \to$ costante).
* **main()**: La funzione principale che orchestra l'intero processo di benchmarking e generazione dei grafici. Definisce le dimensioni dei dataset da testare, chiama la funzione `benchmark_structure` per ciascuna implementazione di coda di priorità (`MaxHeap`, `LinkedList`, `SortedLinkedList`), e poi utilizza le istanze di `PlotGenerator` per produrre una serie di grafici comparativi (tempi per operazione su diverse strutture, tempi per struttura sulle diverse operazioni, benchmark esteso per `MaxHeap`) e grafici di rapporto ($T(n)/\log n$) per validare le complessità asintotiche.

4 Descrizione degli esperimenti condotti e analisi dei risultati sperimentali

4.1 Dati utilizzati

Per le comparazioni principali, sono stati utilizzati i seguenti valori di n:

$$n \in \{1000, 2000, 4000, 8000, 16000\}$$

Queste dimensioni sono state scelte per osservare l'andamento delle prestazioni su input crescenti, includendo sia casi moderati che più estesi.

Data la particolare efficienza del **MaxHeap**, è stato eseguito anche un benchmark separato su scala più ampia:

$$n \in \{1000, 3000, 5000, \ldots, 99000, 100000\} \quad \text{(passi di 2000)}$$

Questo ha permesso di validare sperimentalmente l'andamento logaritmico previsto per le sue operazioni.

I dati inseriti sono interi casuali uniformemente distribuiti nell'intervallo $[1, 10^6]$, per evitare distorsioni legati all'ordine o al contenuto degli elementi.

4.2 Misurazioni

Le misurazioni dei tempi di esecuzione sono state effettuate per ogni struttura, dimensione n e operazione. Il procedimento è il seguente:

- **Ripetizioni multiple**: ogni test è stato ripetuto $r = 3$ volte per ridurre le fluttuazioni casuali e ottenere stime più stabili.
- **Tempo totale per n operazioni**: per ogni struttura e operazione, è stato misurato il tempo complessivo $T_{\text{tot}}(n)$ per eseguire n operazioni consecutive (es. n insert consecutivi su input casuale di lunghezza n).
- **Tempo medio per singola operazione**: per ogni valore di n, il tempo medio è stato calcolato come:

$$T(n) = \frac{1}{r} \sum_{i=1}^{r} \frac{T_{\text{tot},i}(n)}{n} \tag{1}$$

  dove $T_{\text{tot},i}(n)$ è il tempo totale rilevato alla i-esima ripetizione.
- **Strumento di misura**: è stato utilizzato `time.perf_counter()` di Python per una misurazione precisa, catturando il tempo prima e dopo l'esecuzione del blocco di codice da testare.
- **Smussamento tramite media mobile**: per migliorare la leggibilità dei grafici e attenuare eventuali picchi anomali, i dati sono stati filtrati con una media mobile di finestra $w = 3$:

$$T_{\text{sm}}(n) = \frac{1}{w} \sum_{j=0}^{w-1} T(n-j) \tag{2}$$

- **Verifica empirica della complessità**: per validare le complessità attese, sono stati calcolati i rapporti:

$$R(n) = \frac{T(n)}{f(n)} \tag{3}$$

  dove $f(n)$ rappresenta la funzione teorica attesa per l'operazione considerata (es. $f(n) = \log n$ per MaxHeap). Se $R(n)$ tende a una costante, la complessità empirica è coerente con quella teorica.

  Questa analisi è stata applicata in particolare alle operazioni `Insert` ed `Extract Max` su `MaxHeap`, con:

$$R_{\text{Insert}}(n) = \frac{T_{\text{Insert}}(n)}{\log n} \qquad R_{\text{Extract}}(n) = \frac{T_{\text{Extract}}(n)}{\log n} \tag{4}$$

NOTA IMPORTANTE: Il confronto tra le strutture dati è stato eseguito su un range identico di valori di fino a $n = 16\,000$, per numeri più elevati no. In particolare, `LinkedList` e `SortedLinkedList` sono risultati troppo lenti su input di grandi dimensioni, rendendo impraticabile la raccolta di dati significativi oltre una certa soglia (tipicamente $n > 50\,000$).
Sarebbe stato ideale estendere il benchmark anche per `MaxHeap` a valori molto più alti (nell'ordine dei milioni), poiché le complessità asintotiche si riferiscono al comportamento per $n \to \infty$. Tuttavia, per ragioni di tempo e risorse computazionali, ho scelto un compromesso bilanciando accuratezza e fattibilità.

4.3 Risultati sperimentali e commenti analitici

4.3.1 Confronto per struttura dati

Nei grafici per singola struttura (`MaxHeap`, `LinkedList`, `SortedLinkedList`), vediamo il comportamento delle tre operazioni principali: `Insert`, `Maximum` ed `Extract Max`. `MaxHeap` (Figure 4) si distingue per regolarità: le curve di `Insert` e `Extract Max` crescono come previsto in tempi intermedi rispetto a quelli delle altre due strutture dati, mentre `Maximum` resta costante. I grafici sono privi di fluttuazioni rilevanti.

`LinkedList` (Figure 5), al contrario, è molto rapida in `Insert` ($O(1)$), ma inefficiente nelle altre due operazioni, che richiedono una scansione completa della lista ($O(n)$). I tempi variano molto in base alla posizione del massimo, rendendo le curve instabili.

`SortedLinkedList` (Figure 6) consente un `Maximum` e `Extract Max` rapidi (in quanto il massimo corrisponde al primo elemento), ma è peggiore per quanto riguarda l'`Insert`, che è lineare e spesso il più lento tra tutte le strutture. Anche qui i grafici sono meno regolari.

In sintesi: **MaxHeap** è la più solida, mentre **LinkedList** e **SortedLinkedList** soffrono in scenari ampi, specialmente nelle operazioni lineari.

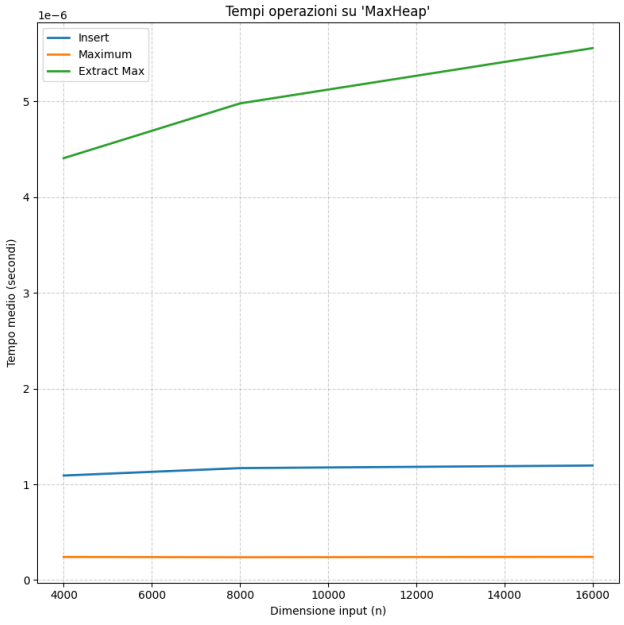

Figure 4: Tempi operazioni su MaxHeap

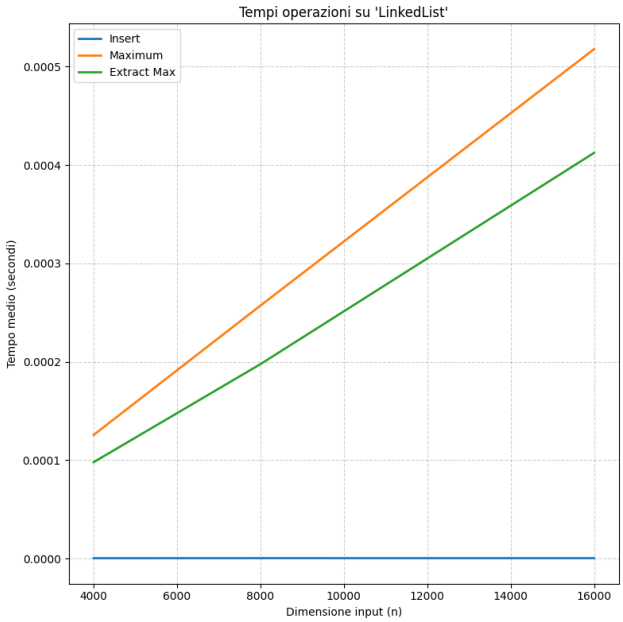

Figure 5: Tempi operazioni su LinkedList

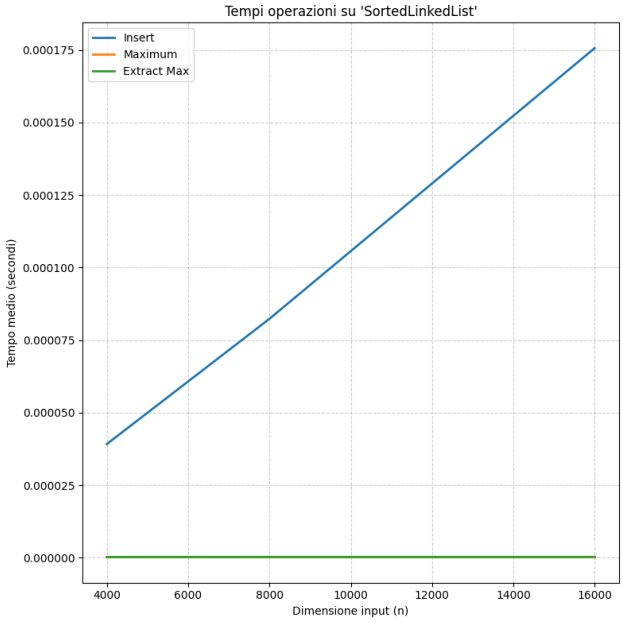

Figure 6: Tempi operazioni su SortedLinkedList

NOTA: In figura 6, le curve delle operazioni `Maximum` (linea arancione) e `Extract Max` (linea verde) risultano sovrapposte. Questo accade perché entrambe le operazioni restituiscono il primo elemento della lista ordinata, con un tempo costante ($O(1)$). Le loro prestazioni coincidono e nel grafico le linee si sovrappongono completamente risultando indistinguibili, se non con uno zoom significativo.

4.3.2 Confronto per operazione

Analizzando i grafici relativi a ciascuna operazione (`Insert`, `Maximum`, `Extract Max`), è possibile confrontare direttamente le prestazioni delle tre strutture dati a parità di compito.

Nel caso di **Insert** (Figure 7), `MaxHeap` si distingue per efficienza e regolarità, confermando la sua complessità $O(\log n)$ e mostrando una buona scalabilità. La `LinkedList`, essendo non ordinata, permette inserimenti istantanei in $O(1)$. Al contrario, `SortedLinkedList` è fortemente penalizzata: l'inserimento richiede un'operazione lineare di ricerca e inserimento ordinato, risultando la più lenta tra le tre.

Per l'operazione **Maximum** (Figure 8), `MaxHeap` e `SortedLinkedList` accedono direttamente al massimo elemento offrendo una prestazione costante, mentre `LinkedList` richiede una scansione completa, quindi con tempi di esecuzione lineari $O(n)$.

Infine nel grafico per l'operazione **Extract Max** (Figure 9), l'efficienza del `MaxHeap` è chiaramente visibile. L'operazione di estrazione del massimo in un `MaxHeap` ha una complessità temporale teorica di $O(\log n)$, dovuta alla necessità di ripristinare la proprietà di heap dopo la rimozione dell'elemento radice. Il grafico mostra che la curva relativa alla `MaxHeap` (linea blu) presenta un andamento che, pur non essendo perfettamente piatto, cresce molto lentamente al crescere di n. Questo comportamento è coerente con una complessità logaritmica. Tuttavia, dato il range di dimensioni dell'input relativamente contenuto (fino a 16000 elementi), l'incremento di tempo è poco più che costante, suggerendo che per questo intervallo di N il termine logaritmico è ancora piccolo e il costo per operazione rimane estremamente basso.

La `SortedLinkedList` mostra un tempo di esecuzione quasi costante e molto basso, in linea con una complessità $O(1)$ per l'estrazione del massimo, poiché l'elemento maggiore è sempre in testa alla lista e la sua rimozione implica solo l'aggiornamento del puntatore alla testa. Questo la rende estremamente efficiente per l'operazione di estrazione del massimo.

Al contrario, la `LinkedList` non ordinata evidenzia le prestazioni peggiori per l'operazione di `Extract Max`. Questa struttura deve prima scorrere l'intera lista per individuare il valore massimo ($O(n)$) e poi eseguire un'ulteriore operazione per rimuoverlo, che anch'essa può richiedere di scorrere la lista ($O(n)$) nel caso peggiore per trovare il nodo precedente. Di conseguenza, il tempo di esecuzione scala linearmente con la dimensione dell'input, mostrando una netta pendenza verso l'alto.

Anche per questa operazione, `MaxHeap` si conferma generalmente la struttura più bilanciata ed efficiente.

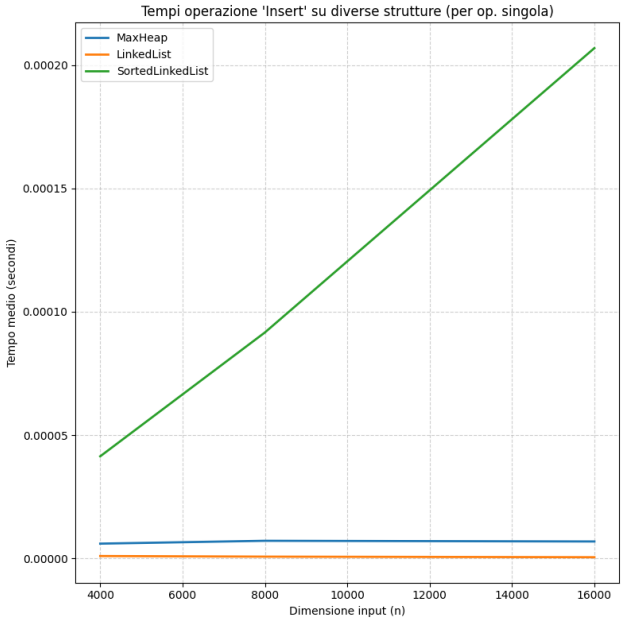

Figure 7: Tempi delle varie strutture dati per Insert

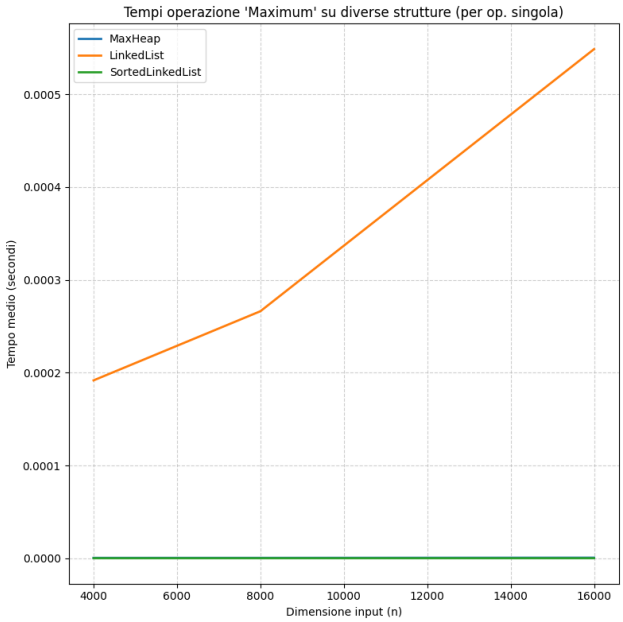

Figure 8: Tempi delle varie strutture dati per Maximum

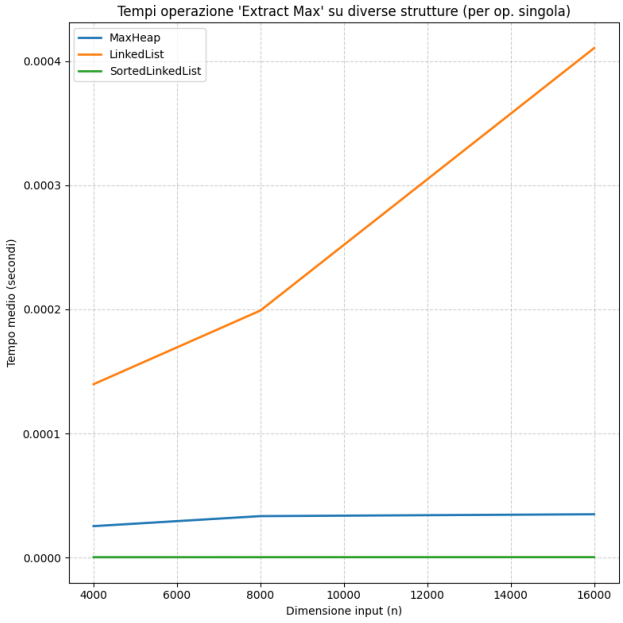

Figure 9: Tempi delle varie strutture dati per Extract Max

NOTA: In figura 8, le curve delle strutture `MaxHeap` (linea blu) e `SortedLinkedList` (linea verde) risultano quasi completamente sovrapposte perché per entrambe le strutture l'operazione di `Maximum` ha una complessità asintotica costante.

4.3.3 Analisi Empirica su MaxHeap

Mentre per strutture come la `LinkedList` e la `SortedLinkedList`, il comportamento asintotico delle operazioni di `Insert` e `Extract Max` (in particolare l'andamento lineare per l'inserimento nella lista ordinata e l'estrazione nella lista non ordinata) era già palese dai primi grafici con un range di dimensioni dell'input più limitato, per la `MaxHeap` la situazione è differente. Le sue complessità teoriche di $O(\log n)$ per le operazioni di inserimento ed estrazione del massimo implicano una crescita dei tempi molto più contenuta rispetto alla complessità lineare ($O(n)$).

Per evidenziare chiaramente questa relazione logaritmica e distinguere l'andamento da un semplice tempo costante per input di piccole dimensioni, è necessaria un'analisi più estesa con dimensioni dell'input significativamente maggiori, raggiungendo fino a $n = 100\,000$.

Per confermare empiricamente la complessità logaritmica di entrambe le operazioni su `MaxHeap`, ovvero che $T(n) \approx C \cdot \log n$ (dove C è una costante), sono stati realizzati grafici che mostrano i rapporti $T(n)/\log n$ per le operazioni `Insert` ed `Extract Max`. Come previsto, tali rapporti tendono a stabilizzarsi verso una costante all'aumentare di n.

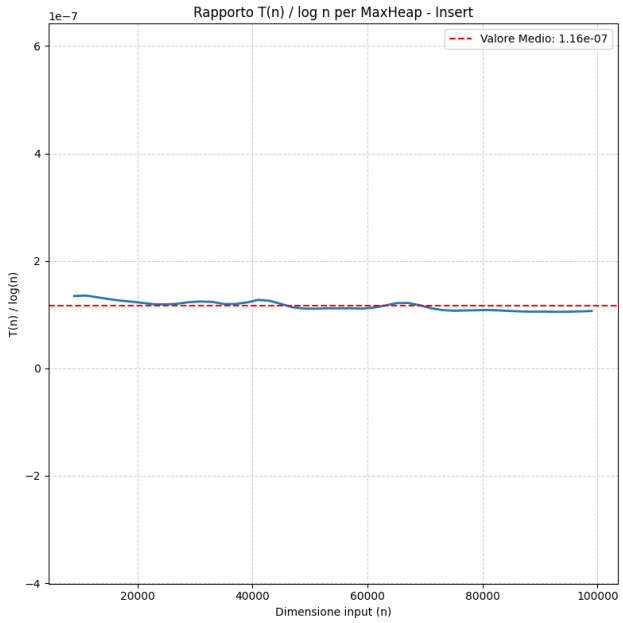

Figure 10: Grafico empirico di MaxHeap per Insert

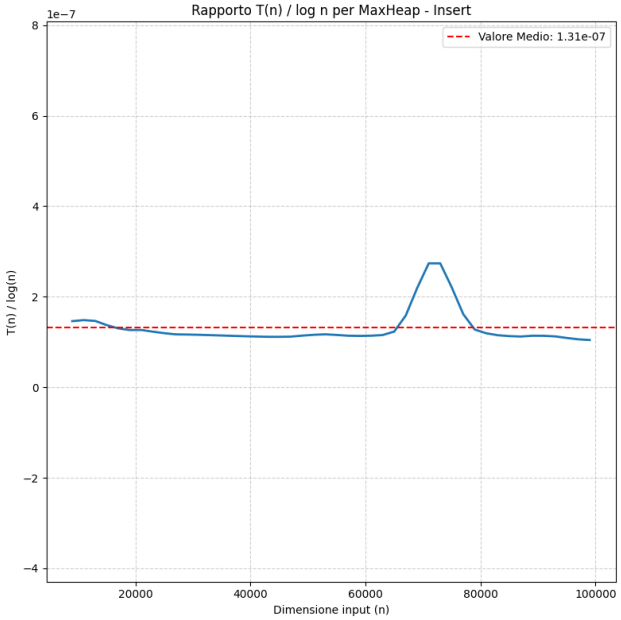

Figure 11: Grafico empirico di MaxHeap per Insert

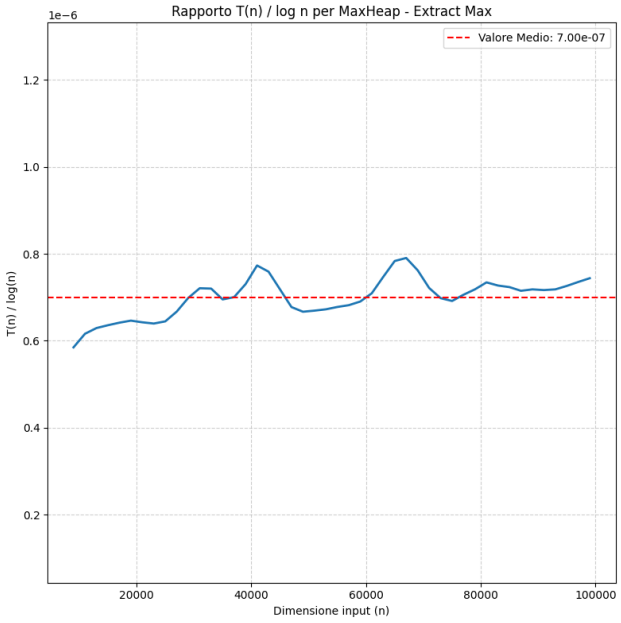

Figure 12: Grafico empirico di MaxHeap per Extract Max

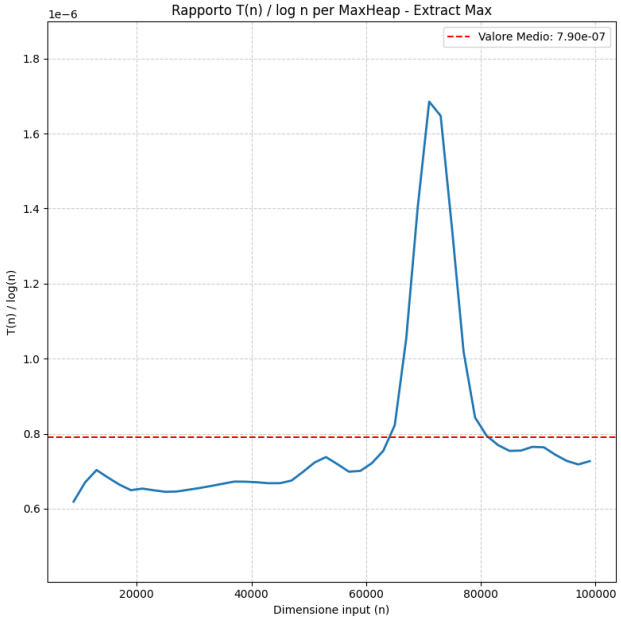

Figure 13: Grafico empirico di MaxHeap per Extract Max

4.3.4 Analisi e Considerazioni sulle Fluttuazioni

Durante l'analisi empirica delle prestazioni, in particolare nei grafici del rapporto $T(n)/\log n$ per le operazioni su `MaxHeap` (Figure 10, 11, 12 e 13), si osserva un andamento non perfettamente costante del rapporto, bensì una serie di piccole oscillazioni, sia verso l'alto che verso il basso, rispetto al valore medio. Questo comportamento non influenza la validità della complessità asintotica $O(\log n)$ (poiché il trend generale è di stabilità intorno a una costante), ma merita un'analisi delle sue possibili cause.

Le fluttuazioni osservate possono essere attribuite a diversi fattori:

- **Natura Randomica dei Dati**: Le prestazioni delle strutture dati sono influenzate dalla specificità dei dati e dall'ordine delle operazioni. Sebbene i dati di input siano generati casualmente per ogni test, la specifica sequenza di elementi per gli n inserti o n estrazioni può introdurre variabilità. Ad esempio, per la `MaxHeap`, un'inserzione che richiede molte risalite nell'albero (`heap_increase_key`) o un'estrazione che necessita di molte discese (`max_heapify`) può momentaneamente aumentare il tempo medio. La formula della media mobile (2) è stata applicata per smussare queste variazioni a breve termine, ma non le elimina completamente.
- **Precisione del Timer e Granularità di Misura**: Pur utilizzando `time.perf_counter()`, che offre un'elevata risoluzione, la misurazione di tempi estremamente brevi (nell'ordine dei micro o nanosecondi, come si vede per le operazioni più veloci) può essere soggetta a un certo "rumore" intrinseco del sistema operativo e dell'hardware. Anche piccole imprecisioni nella temporizzazione si traducono in variazioni relative più significative quando i valori assoluti dei tempi sono molto bassi.
- **Gestione della Memoria e Cache**: La performance di algoritmi e strutture dati è spesso influenzata dall'interazione con la gerarchia di memoria. Accessi alla memoria che causano "cache misses" possono introdurre ritardi imprevedibili. All'aumentare di N, la struttura dati potrebbe superare le dimensioni della cache, portando a cali di performance non lineari ma legati all'architettura hardware.
- **Contesto del Sistema Operativo**: Il sistema operativo esegue numerosi processi in background che possono contendere risorse della CPU e della memoria con il programma di benchmark. Interruzioni, cambi di contesto, e la schedulazione dei processi possono influenzare i tempi di esecuzione in modo non deterministico. Sebbene si cerchi di minimizzare queste interferenze isolando il test, non è possibile eliminarle del tutto in un ambiente non strettamente controllato.

– **Effetti della Normalizzazione nel Rapporto** $T(n)/f(n)$: Il tempo medio per singola operazione $T(n)$ è calcolato come specificato nella formula (1), dividendo il tempo totale per N e mediando sulle ripetizioni. Successivamente, per la verifica della complessità asintotica, si calcola il rapporto $R(n) = T(n)/f(n)$ come in formula (3), e specificamente (4) per la `MaxHeap`. Quando si divide un tempo misurato (che include un certo grado di rumore) per una funzione come $\log n$, che cresce relativamente lentamente, anche piccole variazioni in $T(n)$ possono essere amplificate nel rapporto, specialmente per valori di n a cui $\log n$ è piccolo, causando le oscillazioni osservate.

Nonostante ciò, la tendenza generale dei rapporti è quella di stabilizzarsi attorno a una costante, rendendo valide le considerazioni effettuate sulle complessità asintotiche per le strutture dati analizzate. Le oscillazioni sono quindi interpretate come rumore di misurazione e variabilità intrinseca dell'ambiente di esecuzione, piuttosto che come deviazioni significative.

5 Tesi e Conclusioni Finali

L'analisi condotta sulle tre strutture dati per code con priorità — `MaxHeap`, `LinkedList` e `SortedLinkedList` — ha confermato le complessità teoriche delle operazioni fondamentali: `Insert`, `Maximum` ed `Extract Max`.

MaxHeap si è dimostrata la struttura più efficiente e stabile, con `Insert` ed `Extract Max` che mostrano una crescita logaritmica ($O(\log n)$), come evidenziato dai grafici dei rapporti $T(n)/\log n$ (figure 10, 11, 12 e 13). L'operazione `Maximum`, a tempo costante ($O(1)$), risulta praticamente piatta (figure 8).

SortedLinkedList presenta invece un profilo misto: ottima per `Maximum` ed `Extract Max`, che avvengono in tempo costante grazie alla posizione fissa del massimo (figura 6), ma penalizzata da un `Insert` lineare ($O(n)$), come evidenziato nella figura 7.

LinkedList, non ordinata, risulta efficiente solo per `Insert` ($O(1)$), mentre `Maximum` ed `Extract Max` sono entrambe lineari ($O(n)$), richiedendo una scansione completa della lista (figure 5, 8, 9).

Nel confronto per operazione, `MaxHeap` domina nelle tre categorie per scalabilità e tempi ridotti (figure 7, 8, 9). Le strutture lineari soffrono su input crescenti, con curve più ripide e meno regolari.

In particolare, l'estensione sperimentale su `MaxHeap` fino a $n = 100\,000$ ha permesso di osservare chiaramente l'andamento logaritmico delle operazioni, visibile nella tendenza a valore costante delle linee.

Infine, l'uso di medie mobili e ripetizioni multiple ha garantito misure stabili e affidabili, riducendo l'effetto del rumore e delle fluttuazioni casuali.

L'analisi empirica ha quindi confermato l'ipotesi iniziale per cui **MaxHeap** è la scelta più adatta per gestire code di priorità, soprattutto su dataset di grandi dimensioni, in quanto è in grado di combinare buone prestazioni a stabilità.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein (2009) ”Introduzione agli Algoritmi e Strutture Dati” Terza edizione, McGraw Hill.