Università degli Studi di Firenze
Dipartimento di Ingegneria dell'Informazione

Heap - Lista concatenata - Lista concatenata ordinata

Autore:
Matteo Gerlotti

N° Matricola:
7025024

Corso principale:
Algoritmi e Strutture Dati

Docente corso:
Simone Marinai

Contents

1 Introduzione

1.1 Presentazione dell'esercizio

L'obiettivo di questo elaborato è analizzare, implementare e confrontare tre differenti strutture dati utilizzabili per realizzare una *coda di priorità*:

- **Heap binario**
- **Lista concatenata non ordinata**
- **Lista concatenata ordinata (in senso decrescente)**

Tali strutture rappresentano tre approcci diversi alla gestione dell'operazione *extract_max*, che costituisce il fulcro funzionale della coda di priorità.

Il lavoro si articola in tre fasi principali:

- **implementazione** delle tre strutture dati in linguaggio Python, partendo da un'interfaccia comune;
- **analisi teorica** delle loro prestazioni, con particolare attenzione ai costi asintotici delle operazioni fondamentali;
- **valutazione sperimentale**, tramite misurazioni del tempo di esecuzione su diversi insiemi di dati in input.

Lo scopo finale è mettere in evidenza punti di forza, limiti e casi d'uso preferibili per ciascuna implementazione. La relazione include:

- descrizione delle strutture dati analizzate;
- motivazioni delle scelte implementative;
- metodologia di misura dei tempi di esecuzione;
- confronto tramite esperimenti delle prestazioni delle tre implementazioni;
- analisi dei risultati in relazione alla complessità teorica prevista;
- discussione vantaggi/svantaggi e conclusioni.

1.2 Specifiche della piattaforma di test

Per ciascun esperimento che verrà presentato ho usato la stessa piattaforma di test. Partendo dall'hardware e dal sistema operativo del computer:

- **CPU** : `1,4 GHz Intel Core i5 quad-core`
- **RAM** : `8 GB 2133 MHz LPDDR3`
- **GPU** : `Intel Iris Plus Graphics 645 (1536 MB)`
- **SSD** : `Macintosh HD 250 GB`
- **OS** : `Sequoia 15.7.2`

Il progetto è realizzato con il linguaggio di programmazione Python e il codice è stato sviluppato nell'IDE **PyCharm 2023.3.7**.

Per questa relazione in LaTeX ho utilizzato l'editor online **Overleaf**.

2 Descrizione del problema

Una **coda di priorità** è una struttura dati che permette di memorizzare elementi associati a una priorità e di estrarre rapidamente l'elemento con priorità massima.

Le operazioni fondamentali sono:

- **insert(key)**: inserisce un valore (*key*) nella struttura;
- **peek()**: restituisce il valore massimo senza rimuoverlo;
- **extract_max()**: rimuove e restituisce il valore massimo;
- **size()**: restituisce il numero di elementi presenti.

Nel progetto vengono analizzate tre differenti strategie implementative:

1. **Heap binario**, struttura ad albero bilanciato che consente tempi logaritmici sia per l'inserimento sia per l'estrazione del massimo.
2. **Lista concatenata non ordinata**, che permette inserimenti molto efficienti, ma richiede una scansione completa per individuare l'elemento massimo.
3. **Lista concatenata ordinata**, che mantiene gli elementi in ordine decrescente, garantendo estrazioni immediate ma inserimenti più costosi.

Il problema affrontato consiste dunque nel *confrontare* queste tre implementazioni sia dal punto di vista teorico, tramite l'analisi della complessità computazionale delle operazioni fondamentali, sia dal punto di vista pratico, attraverso esperimenti basati su insiemi di input eterogenei e misurazioni reali dei tempi di esecuzione.

L'obiettivo finale è determinare quale delle tre strutture risulta più efficiente in diversi scenari operativi e confermare, tramite evidenze sperimentali, le previsioni teoriche derivate dall'analisi algoritmica.

3 Strutture dati e algoritmi

In questa sezione vengono descritte le tre strutture dati implementate per la realizzazione della coda di priorità, insieme ai relativi algoritmi per le operazioni fondamentali: *insert*, *extract_max* e *peek*. Per ciascuna struttura vengono inoltre discusse le complessità attese.

3.1 Heap binario

L'heap binario utilizzato in questo progetto è un **max-heap**, ovvero una struttura ad albero binario completo in cui ogni nodo è maggiore o uguale ai propri figli. L'heap è rappresentato tramite un array, secondo l'implementazione standard:

- il nodo all'indice i ha come figli gli indici $2i + 1$ e $2i + 2$;
- il padre del nodo all'indice i si trova in $\lfloor (i - 1)/2 \rfloor$.

Algoritmi principali

- **insert(key)**: la chiave (*key*) viene aggiunta in fondo all'array e riportata verso l'alto tramite l'operazione *heapify_up*, finché la proprietà di heap non è ristabilita.
- **extract_max()**: il massimo, situato in radice, viene scambiato con l'ultimo elemento; l'ultimo elemento viene rimosso e l'heap viene riordinato tramite *heapify_down*.
- **peek()**: restituisce il valore in radice, che rappresenta il massimo.

Complessità attesa

- insert: $\mathcal{O}(\log n)$
- extract_max: $\mathcal{O}(\log n)$
- peek: $\mathcal{O}(1)$

3.2 Lista concatenata non ordinata

La lista concatenata non ordinata è una struttura composta da nodi collegati tramite puntatori. Ogni nodo contiene una chiave e un riferimento al nodo successivo. L'inserimento avviene sempre **in testa**, senza alcuna operazione di ordinamento.

Algoritmi principali

- **insert(key)**: il nuovo nodo viene inserito in testa in tempo costante.
- **extract_max()**: prevede due passaggi: una prima scansione per identificare il massimo, e una seconda per rimuoverlo.
- **peek()**: richiede una scansione completa per individuare la chiave massima.

Complessità attesa

- insert: $\mathcal{O}(1)$
- extract_max: $\mathcal{O}(n)$
- peek: $\mathcal{O}(n)$

Questa implementazione è efficiente quando il numero di inserimenti è molto maggiore rispetto alle estrazioni del massimo.

3.3 Lista concatenata ordinata

In questo caso la lista viene mantenuta sempre in **ordine decrescente**: il nodo in testa è il massimo. Ciò consente di ottenere estrazioni rapide.

Algoritmi principali

- **insert(key)**: la nuova chiave viene inserita nella posizione corretta per mantenere l'ordine; nel caso peggiore, l'intera lista deve essere attraversata.
- **extract_max()**: consiste semplicemente nella rimozione del primo nodo.
- **peek()**: il massimo è in testa e viene restituito in tempo costante.

Complessità attesa

- insert: $\mathcal{O}(n)$
- extract_max: $\mathcal{O}(1)$
- peek: $\mathcal{O}(1)$

Questa struttura è ideale nei casi in cui l'operazione dominante è *extract_max*, mentre il costo dell'inserimento risulta trascurabile.

4 Progettazione e struttura del codice

In questa sezione viene descritta l'organizzazione del progetto, la struttura dei file, il ruolo delle principali classi e le motivazioni delle scelte implementative adottate.

4.1 Diagramma UML

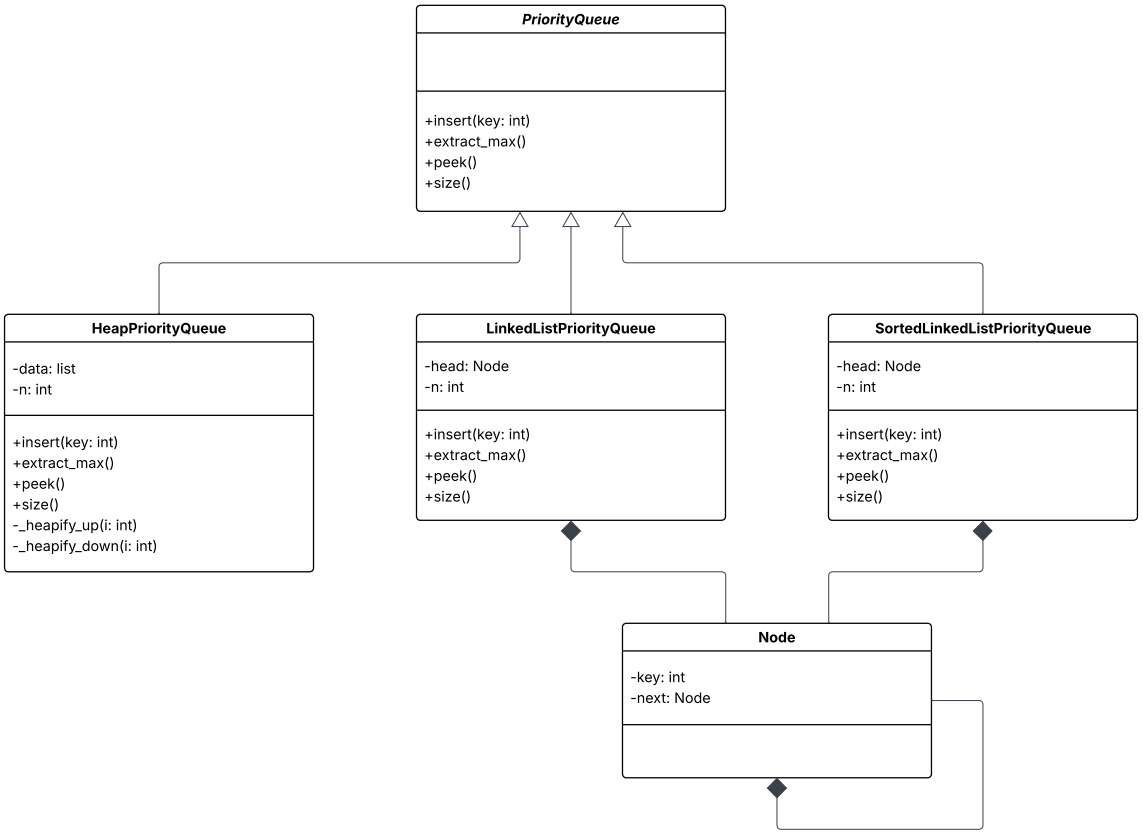

Figure 1: Diagramma UML delle classi implementate.

Il diagramma 1 evidenzia l'utilizzo di una classe astratta comune e la separazione chiara tra le diverse strategie implementative.

4.2 Struttura dei file del progetto

- `priority_queue_base.py`
  contiene l'interfaccia comune `PriorityQueue`, che definisce le operazioni essenziali.
- `heap_priority_queue.py`
  implementazione della coda di priorità tramite heap binario.
- `linked_list_priority_queue.py`
  implementazione basata su una lista concatenata non ordinata.
- `sorted_linked_list_priority_queue.py`
  implementazione basata su una lista concatenata mantenuta in ordine decrescente.
- `utils.py`
  contiene funzioni ausiliarie per la generazione degli input, il calcolo dei tempi e la verifica dei risultati.
- `tests.py`
  script principale che esegue i test sperimentali, misura i tempi e salva i risultati in formato CSV.
- `plot_results.py`
  genera grafici comparativi delle prestazioni usando i dati aggregati.

Interfaccia comune: PriorityQueue

La classe `PriorityQueue` funge da interfaccia ed espone i metodi fondamentali che ogni implementazione deve fornire:

- **insert(key)**
- **extract_max()**
- **peek()**
- **size()**

Questa scelta permette di utilizzare le diverse implementazioni in modo intercambiabile all'interno dello script di test, garantendo modularità ed estensibilità del progetto.

4.3 Analisi delle scelte implementative

4.3.1 Separazione in moduli distinti

La suddivisione in più file consente una gestione ordinata del progetto: ogni struttura dati ha un file dedicato, rendendo il codice più leggibile, modificabile e testabile separatamente.

4.3.2 Uso di una lista collegata per le implementazioni

Non usando la struttura `list` di Python come implementazione diretta della lista concatenata, le due liste sono costruite tramite una classe `Node`, che dispone di:

- un campo `key` (valore);
- un puntatore `next` al nodo successivo.

4.3.3 Heap rappresentato tramite array

La rappresentazione tramite array è la più efficiente e standard per un heap binario, riducendo l'overhead dei puntatori (costo aggiuntivo in memoria e tempo di esecuzione) e semplificando l'accesso ai figli e al padre.

4.3.4 Strutture dati compatibili con i test

Le tre classi sono progettate affinché:

- abbiano gli stessi metodi pubblici (compatibilità con `PriorityQueue`);
- possano essere passate allo script `tests.py` senza differenze;
- producano lo stesso comportamento logico (*max-heap*).

4.4 Descrizione dei metodi implementati

In questa sezione vengono descritti i metodi fondamentali delle tre implementazioni della coda di priorità. Le tre strutture si differenziano principalmente per la gestione interna dei dati e per il costo computazionale delle operazioni.

Metodo insert(key) : inserisce un nuovo elemento nella coda di priorità.

- **HeapPriorityQueue**: l'elemento viene inserito in fondo all'array e quindi "risale" tramite la procedura *heapify_up*. Questo garantisce il mantenimento della proprietà di heap.
  Complessità: $\mathcal{O}(\log n)$ nel caso medio e peggiore.
- **LinkedListPriorityQueue**: il nodo viene inserito sempre in testa, senza confronti né scansioni.
  Complessità: $\mathcal{O}(1)$.
- **SortedLinkedListPriorityQueue**: viene cercata la posizione corretta affinché la lista rimanga ordinata in senso decrescente. Inserire richiede quindi di scorrere la lista fino al punto appropriato.
  Complessità: $\mathcal{O}(n)$.

Metodo extract_max() : rimuove e restituisce l'elemento con valore massimo.

- **HeapPriorityQueue**: il massimo è sempre in posizione 0 dell'array. Dopo la rimozione, l'ultimo elemento viene spostato in testa e si applica *heapify_down* per ripristinare la struttura dell'heap.
  Complessità: $\mathcal{O}(\log n)$.
- **LinkedListPriorityQueue**: occorre prima cercare il massimo scansionando l'intera lista, poi rimuovere il nodo corrispondente tramite gestione dei puntatori.
  Complessità: $\mathcal{O}(n)$.
- **SortedLinkedListPriorityQueue**: il massimo è sempre in testa, quindi basta rimuovere il primo nodo.
  Complessità: $\mathcal{O}(1)$.

Metodo peek() : restituisce il valore massimo senza rimuoverlo.

- **HeapPriorityQueue**: il massimo è l'elemento in testa all'array.
  Complessità: $\mathcal{O}(1)$.
- **LinkedListPriorityQueue**: è necessario scorrere l'intera lista per trovare il massimo.
  Complessità: $\mathcal{O}(n)$.
- **SortedLinkedListPriorityQueue**: il massimo è sempre `head.key`.
  Complessità: $\mathcal{O}(1)$.

Metodo size() : restituisce il numero di elementi presenti nella struttura.

In tutte le implementazioni tale informazione è mantenuta come contatore interno aggiornato ad ogni operazione. Complessità per tutte le implementazioni: $\mathcal{O}(1)$.

Metodi interni

Ogni implementazione include metodi ausiliari non esposti all'esterno:

- **HeapPriorityQueue**: `_heapify_up()`, `_heapify_down()`, `_swap()`, `_parent()`, `_left()`, `_right()`.
- **LinkedListPriorityQueue**: nessun metodo interno particolarmente complesso, gestisce i nodi tramite due passaggi (ricerca e rimozione).
- **SortedLinkedListPriorityQueue**: logica di ricerca della posizione corretta per l'inserimento.

Questi metodi interni non fanno parte dell'interfaccia pubblica ma sono fondamentali per garantire la correttezza e l'efficienza delle rispettive strutture dati.

4.5 Funzioni di supporto

Il file `utils.py` contiene:

- funzioni per generare input di diversa natura (*random, ascending, descending, repeated*);
- un misuratore dei tempi `time_function()`;
- funzioni per aggregare i tempi e produrre statistiche;
- una funzione per verificare la correttezza dell'estrazione del massimo.

4.6 Script per i test e per i grafici

Lo script `tests.py` esegue:

- tutte le configurazioni di input;
- più ripetizioni degli stessi test;
- salvataggio dei dati grezzi e aggregati in CSV;
- controllo della correttezza dell'estrazione.

Lo script `plot_results.py` utilizza i dati aggregati per produrre grafici comparativi dei tempi di:

- **insert**;
- **extract_all**.

Le immagini prodotte vengono salvate nella cartella `results/plots/`.

5 Valutazione a priori delle prestazioni

Prima di effettuare gli esperimenti è utile analizzare teoricamente il comportamento delle tre strutture dati, valutando il costo computazionale delle operazioni fondamentali. Questa fase permette di formulare aspettative precise sulle prestazioni, che saranno successivamente verificate attraverso misure sperimentali.

In questa sezione vengono confrontati:

- i costi asintotici delle operazioni *insert*, *extract_max* e *peek*;
- i vantaggi e gli svantaggi strutturali di ciascuna implementazione;
- gli scenari applicativi per cui ogni struttura risulta preferibile.

5.1 Confronto delle complessità

La Tabella 1 riassume le complessità attese delle operazioni fondamentali:

Implementazione	**insert**	**extract_max**	**peek**
Heap binario	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
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 asintotiche delle tre implementazioni.

5.2 Interpretazione delle complessità

Le differenze tra le strutture analizzate derivano direttamente dal modo in cui sono organizzati i dati:

- Nell'**heap binario** la proprietà di heap permette accesso immediato al massimo e inserimenti/estrazioni logaritmiche. È generalmente considerato l'approccio più bilanciato.
- La **lista concatenata non ordinata** favorisce gli inserimenti, ma è inefficiente quando si cerca il massimo, poiché richiede una scansione completa.
- La **lista concatenata ordinata** massimizza l'efficienza per *peek* ed *extract_max*, a costo di inserimenti più costosi.

5.3 Scenari applicativi attesi

Sulla base della teoria, possiamo prevedere:

- Se **gli inserimenti sono molto più frequenti delle estrazioni**, la lista non ordinata risulta vantaggiosa.
- Se **le estrazioni sono l'operazione dominante**, la lista ordinata è la scelta più efficiente.
- Se si desidera una struttura **equilibrata e scalabile**, l'heap binario rappresenta il compromesso migliore.

5.4 Comportamento atteso per input di diversa natura

I test verranno eseguiti su quattro tipologie di input:

- *random*: distribuzione uniforme;
- *ascending*: sequenza crescente;
- *descending*: sequenza decrescente;
- *repeated*: valori ripetuti in un range ristretto.

Sulla base delle complessità:

- L'**heap** dovrebbe essere relativamente insensibile alla natura dell'input, poiché gli inserimenti rispettano sempre la stessa logica.
- La **lista non ordinata** non ha alcuna dipendenza dall'ordine dell'input, essendo sempre *insert-in-head*.
- La **lista ordinata** presenterà casi peggiori per input ascendenti (ogni valore viene inserito in fondo).

5.5 Aspettative sperimentali

Sulla base dell'analisi teorica, ci aspettiamo che:

- l'heap risulti la migliore struttura nel caso generale e per input di grandi dimensioni;
- la lista non ordinata sia un'opzione valida solo per scenari in cui si effettuano molti inserimenti, mentre le estrazioni sono rare;
- la lista ordinata sia ottima nei test di *extract_max*, ma mostri tempi di *insert* nettamente superiori;

Queste previsioni saranno confermate o eventualmente smentite attraverso gli esperimenti discussi nelle sezioni successive.

6 Esperimenti e metodologia

In questa sezione vengono descritti la strategia sperimentale, le modalità con cui sono stati generati gli input di test e la configurazione utilizzata per misurare le prestazioni delle tre implementazioni della coda di priorità. L'obiettivo è ottenere misurazioni affidabili, confrontabili e riproducibili.

6.1 Obiettivi dell'esperimento

Gli esperimenti mirano a:

- misurare il tempo totale di esecuzione dell'operazione **insert** su n elementi;
- misurare il tempo totale di esecuzione dell'operazione **extract_all** dopo gli inserimenti;
- valutare l'impatto della dimensione di input e della sua distribuzione;
- confrontare le prestazioni reali con le complessità teoriche attese;
- verificare la correttezza delle estrazioni (*extract_max*) tramite un controllo automatico.

I test sono stati eseguiti tramite lo script `tests.py`, che automatizza l'intero processo e salva i risultati in formato CSV.

6.2 Generazione degli input

La generazione dei dati avviene tramite la funzione `generate_input()` contenuta in `utils.py`. Sono stati considerati quattro tipi di input:

- **random**: valori casuali uniformi in un intervallo;
- **ascending**: sequenza crescente, dal più piccolo al più grande;
- **descending**: sequenza decrescente;
- **repeated**: valori ripetuti presi da un intervallo ristretto.

Queste tipologie permettono di osservare anche i casi in cui un'implementazione può incorrere in comportamenti particolarmente favorevoli o sfavorevoli.

6.3 Misurazione dei tempi

La misurazione dei tempi è stata effettuata con la funzione `time.perf_counter()`, che garantisce alta precisione. Ogni configurazione è stata eseguita più volte per ridurre la variabilità dovuta al sistema operativo o al carico macchina.

Per ogni esecuzione vengono registrati:

- tempo totale dell'operazione **insert**;
- tempo totale dell'operazione **extract_all**;
- verifica della correttezza dell'estrazione;
- valori salvati in un file CSV (`raw_results.csv`).

Successivamente, i risultati vengono aggregati in:

- **mediana**,
- **media**,
- **deviazione standard**,

e salvati nel file `aggregated_results.csv` tramite la funzione `aggregate_times()`.

6.4 Esecuzione dei test: configurazione di default

Una prima esecuzione "veloce" dei test può essere effettuata senza specificare parametri aggiuntivi:

```
python tests.py
```

Questa configurazione esegue:

- dimensioni di input: 100, 500, 1000;
- 5 ripetizioni per ogni configurazione;
- tutti e quattro i tipi di input;
- le tre implementazioni analizzate.

Questa modalità è utile per:

- verificare il corretto funzionamento del programma;
- ottenere risultati preliminari;
- assicurarsi che non ci siano errori nelle strutture dati.

6.5 Esecuzione dei test completi

Per la valutazione approfondita delle prestazioni è stato utilizzato un set esteso di parametri:

```
python tests.py --ns 1000,3000,5000,10000
                --runs 7
                --cases random,ascending,descending,repeated
```

Questa configurazione genera in totale:

4 dimensioni $\times$ 4 tipi di input $\times$ 3 implementazioni $\times$ 7 ripetizioni = 336 esecuzioni

I vantaggi di questa configurazione completa sono:

- una migliore stabilizzazione statistica dei risultati;
- possibilità di osservare chiaramente l'andamento asintotico;
- capacità di distinguere i comportamenti in scenari diversi (ordinato, ripetuto, casuale);
- produzione di grafici più attendibili e regolari.

I risultati raccolti in questa modalità sono quelli utilizzati per l'analisi finale presentata nella sezione successiva.

6.6 Generazione dei grafici

I dati aggregati vengono elaborati tramite lo script:

```
python plot_results.py
```

che produce grafici comparativi (con `matplotlib`) salvati nella cartella:

```
results/plots/
```

Sono generati grafici separati per:

- **insert**
- **extract_all**

e per ognuno dei quattro tipi di input.

I grafici permettono un confronto visivo immediato dell'andamento delle curve e confermano (o smentiscono) le aspettative teoriche presentate nella sezione precedente.

7 Risultati sperimentali

Qui vengono presentati i risultati ottenuti eseguendo i test descritti nella sezione precedente. I tempi di esecuzione sono stati registrati per le operazioni `insert` ed `extract_all` su insiemi di input di diversa dimensione e ordine.

Le figure riportate mostrano, per ogni tipo di input, l'andamento dei tempi al crescere della dimensione n e permettono un confronto diretto tra le tre implementazioni della coda di priorità.

7.1 Tempi di esecuzione per l'operazione `insert`

I grafici relativi all'operazione `insert` mostrano il tempo totale necessario per inserire n elementi nella struttura. Le misure sono state effettuate per i quattro tipi di input: **random**, **ascending**, **descending**, **repeated**

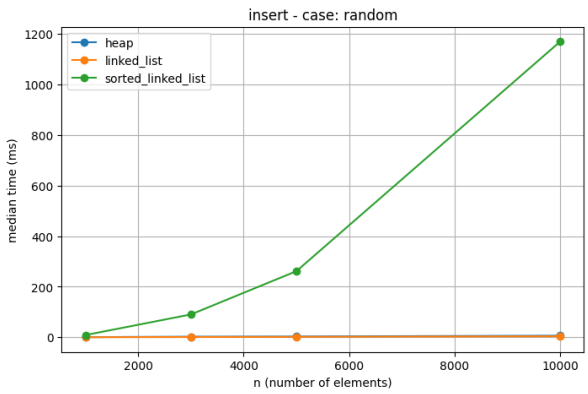

Figure 2: Input casuale (*random*).

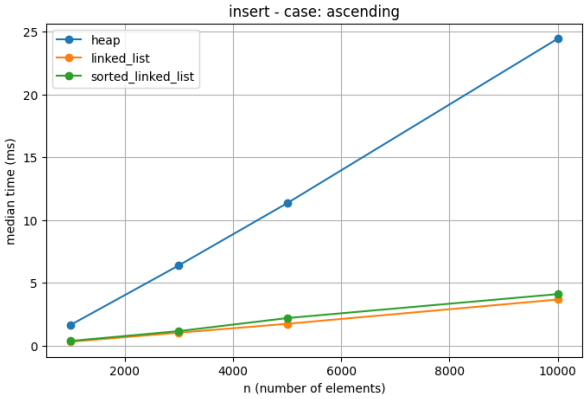

Figure 3: Input crescente (*ascending*).

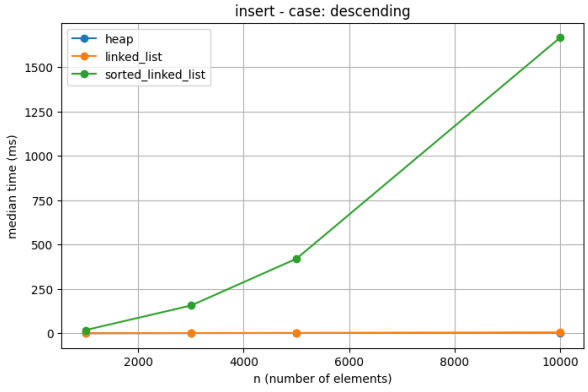

Figure 4: Input decrescente (*descending*).

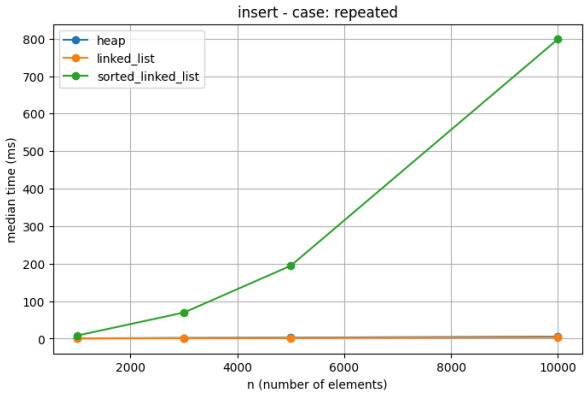

Figure 5: Valori ripetuti (*repeated*).

7.2 Tempi di esecuzione per l'operazione `extract_all`

Misure dei tempi necessari per estrarre tutti gli elementi dopo aver riempito la struttura. Ogni estrazione esegue ripetutamente l'operazione `extract_max`, fino allo svuotamento. Anche qui vengono riportati i risultati per tutte le tipologie di input considerate.

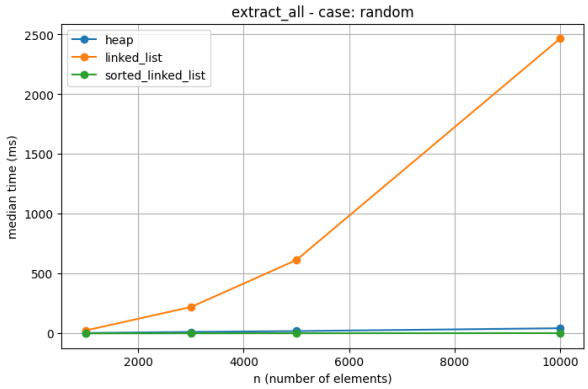

Figure 6: Input casuale (*random*).

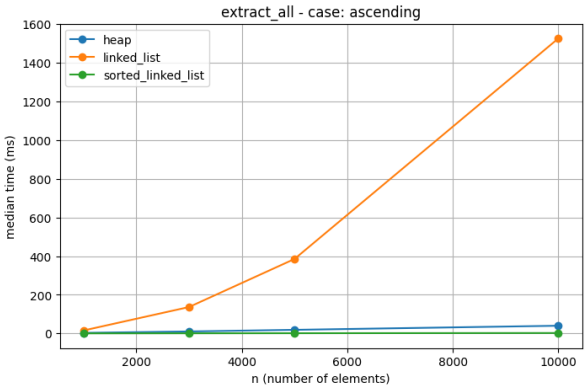

Figure 7: Input crescente (*ascending*).

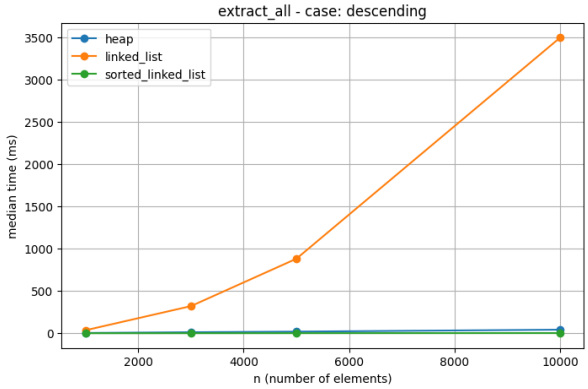

Figure 8: Input decrescente (*descending*).

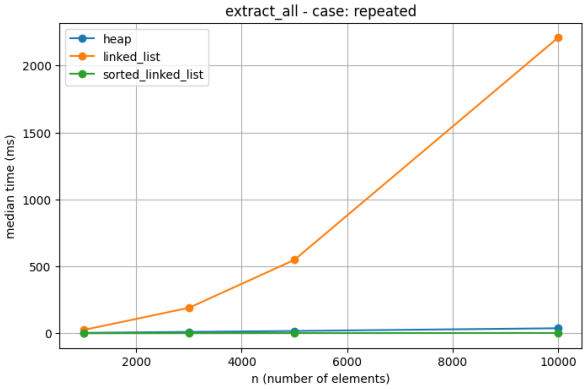

Figure 9: Valori ripetuti (*repeated*).

7.3 Tabelle riassuntive

Per facilitare la consultazione e rendere più immediato il confronto, sono riportate tabelle riassuntive contenenti i tempi mediani (caso *random*) per ciascuna implementazione al variare di n. (La mediana è meno sensibile alla presenza di valori anomali rispetto alla media e fornisce una stima più robusta del comportamento reale dell'algoritmo).

7.3.1 inserimento (Figure 2)

Implementazione	**n=1000**	**n=3000**	**n=5000**	**n=10000**
Heap	0.569	2.474	3.118	6.435
Lista non ordinata	0.352	1.123	1.782	4.092
Lista ordinata	8.848	90.465	261.675	1170.218

Table 2: Tabella riassuntiva per l'operazione `insert` (tempi in **ms**).

7.3.2 estrazione completa (Figure 6)

Implementazione	**n=1000**	**n=3000**	**n=5000**	**n=10000**
Heap	2.925	12.257	19.403	42.980
Lista non ordinata	24.124	220.558	613.597	2466.196
Lista ordinata	0.214	0.734	1.081	2.217

Table 3: Tabella riassuntiva per l'operazione `extract_all` (tempi in **ms**).

8 Analisi dei risultati

L'analisi dei risultati sperimentali conferma le previsioni teoriche esposte nelle sezioni precedenti. In particolare, il confronto tra le tre implementazioni mette in evidenza differenze significative nei tempi delle operazioni `insert` ed `extract_all`, soprattutto al crescere della dimensione dell'input.

Comportamento dell'operazione `insert`

L'**heap binario** mostra una crescita del tempo in linea con la complessità attesa $\mathcal{O}(\log n)$: l'aumento dei tempi è moderato e regolare per tutti i tipi di input. La **lista non ordinata** risulta la più efficiente nell'inserimento, confermando la complessità costante $\mathcal{O}(1)$: i tempi rimangono molto bassi anche per n elevati. La **lista ordinata** è invece nettamente la più lenta, con tempi che crescono linearmente secondo $\mathcal{O}(n)$. L'effetto è particolarmente evidente con liste decrescenti, dove ogni nuovo elemento viene inserito in coda.

Comportamento dell'operazione `extract_all`

L'operazione di estrazione evidenzia un comportamento opposto. La **lista non ordinata** risulta la più inefficiente: per ogni estrazione è necessario scandire l'intera lista, con un costo totale che cresce in modo quadratico rispetto a n, rendendo l'implementazione inadatta per scenari con molte estrazioni. La **lista ordinata**, al contrario, è la più veloce: l'elemento massimo si trova sempre in testa e l'operazione richiede tempo costante $\mathcal{O}(1)$; i grafici mostrano valori pressoché invariati al crescere di n. L'**heap** rappresenta un buon compromesso: il tempo complessivo cresce quasi linearmente, in accordo con la complessità $\mathcal{O}(n \log n)$ necessaria per estrarre tutti gli elementi.

Influenza del tipo di input

I test indicano che il tipo di input influenza soprattutto l'heap, che mostra tempi leggermente maggiori per sequenze crescenti o decrescenti a causa della struttura dell'albero dopo successive inserzioni. Le due liste mostrano invece un comportamento quasi identico sui diversi casi, poiché la loro complessità dipende unicamente dalla lunghezza della struttura.

Sintesi

Nel complesso:

- la lista non ordinata eccelle negli inserimenti ma è molto inefficiente nelle estrazioni;
- la lista ordinata è eccellente nelle estrazioni ma lenta negli inserimenti;
- l'heap si conferma il miglior compromesso, con prestazioni stabili e scalabili.

Queste osservazioni trovano riscontro coerente sia nei grafici prodotti sia nei valori numerici estratti dai file CSV.

9 Conclusioni

L'obiettivo di questo elaborato era confrontare tre implementazioni di una coda di priorità sia dal punto di vista teorico sia mediante un'analisi sperimentale.
I risultati ottenuti confermano pienamente le previsioni basate sulla complessità delle operazioni fondamentali.

La **lista non ordinata** si è dimostrata estremamente efficiente nell'operazione di `insert`, grazie al costo costante $\mathcal{O}(1)$, ma molto lenta nell'estrazione del massimo, dove è necessario scandire l'intera struttura. Questa implementazione è adatta solo a scenari con molte inserzioni e poche estrazioni.

La **lista ordinata** presenta il comportamento opposto: inserire un elemento richiede tempo lineare, ma l'estrazione del massimo è immediata. È quindi ideale per applicazioni in cui l'operazione dominante è `extract_max`.

L'**heap binario** rappresenta il miglior compromesso: offre tempi logaritmici sia per l'inserimento sia per l'estrazione e prestazioni stabili al crescere di n. I risultati sperimentali mostrano una buona scalabilità e un comportamento prevedibile, indipendentemente dal tipo di input.

In conclusione, nessuna delle tre strutture è universalmente migliore: la scelta dipende dal carico di lavoro e dal numero relativo di inserimenti ed estrazioni. Tuttavia, l'heap risulta nella maggior parte dei casi la soluzione più equilibrata, combinando efficienza, robustezza e semplicità implementativa.