Esercizio A - Coda con priorità

Andrea Delli (matricola 7052940)

Marzo/Aprile 2023

Indice

1 Introduzione

Questa relazione ha l'obiettivo di confrontare tre diverse implementazioni di una coda con priorità, realizzate attraverso diverse strutture dati, con lo scopo di confrontarne le differenze implementative e le prestazioni.

Le strutture dati considerate sono la **lista concatenata**, la **lista concatenata ordinata** e l'**heap**.

1.1 Ambiente di testing

Il codice per eseguire i test è stato realizzato in Python, ed eseguito utilizzando l'interprete Python versione 3.11.0.

L'ambiente di sviluppo utilizzato è l'IDE PyCharm 2022.3, su un laptop Aspire 7, con un processore Intel Core i7-9750H 2.6HGz, 24GB di RAM, con sistema operativo Windows 11 Home versione 22H2.

La seguente relazione è stata scritta in LaTeXtramite il web editor Overleaf. Il diagramma delle classi in Figura 1 è stato realizzato tramite il software Lucidchart.

1.2 Librerie utilizzate

Il programma Python creato utilizza i seguenti moduli:

- `timeit`: misurazione dei tempi di esecuzione degli algoritmi

  (funzione `timeit.timeit(stmt, number)`)

- `numpy`: generazione di array con valori casuali

  (funzione `numpy.random.randint()`)

- `matplotlib`: generazione dei grafici e delle tabelle, per analizzare visivamente le prestazioni dei due algoritmi

  (funzioni `matplotlib.pyplot.plot()` e `matplotlib.pyplot.table()`)

- `scipy`: rappresentazione migliore dei grafici, per inserire curve che rappresentano l'andamento previsto dei dati

  (funzione `scipy.interpolate.interp1d()`)

- `os`: creazione delle cartelle dove salvare le immagini di grafici e tabelle

  (funzioni `os.path.exists(...)` e `os.makedirs(...)`)

2 Cenni teorici

Una **coda di priorità** è una struttura dati che contiene un insieme dinamico di elementi, a ognuno dei quali è associato un valore che ne identifica la priorità.

In alcuni contesti, ad esempio nello scheduling di alcuni processori, un valore basso di priorità indica una priorità più alta.
In altri contesti invece un valore più alto indica una priorità alta, approccio usato anche ai fini di questa relazione.

In una implementazione di una coda con priorità è necessario inserire alcune funzioni per poterla utilizzare. In particolare, è necessario poter inserire dei valori nella coda, che avranno una loro priorità, e sarà necessario poter trovare ed estrarre il valore a priorità più alta.

2.1 Struttura del codice

La struttura del codice è rappresentata in Figura 1 tramite l'utilizzo di un diagramma delle classi.
L'obiettivo è quello di realizzare una coda con priorità utilizzando diverse strutture dati. Per questo motivo, è stata realizzata una classe astratta **Queue**, che indica i metodi necessari per far funzionare correttamente una coda con priorità. In particolare, ci sono le firme dei metodi per l'inserimento di un valore nella coda, per la ricerca e per l'estrazione del valore massimo in coda.
Le classi derivate da Queue sono classi concrete, che implementano i metodi richiesti in modo tale da rendere utilizzabile la coda con priorità.

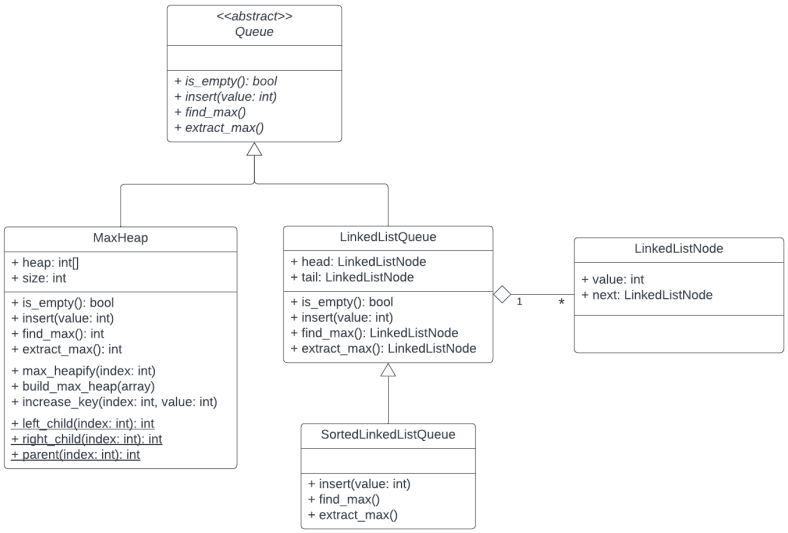

Figura 1: Diagramma delle classi

2.2 Lista concatenata

Una lista concatenata è una collezione di elementi in cui l'ordine dei dati non dipende loro dalla disposizione in memoria, bensì ogni elemento ha un riferimento all'elemento successivo. La struttura dati consiste quindi in una sequenza di nodi, ognuno delle quali contiene un dato e una referenza al nodo successivo, come rappresentato in Figura 2.

Nel caso di implementazione di una coda con priorità tramite una lista concatenata, il dato è un valore intero che rappresenta la priorità del nodo.

La lista concatenata realizzata (classe **LinkedList** in Figura 1) implementa i metodi necessari al funzionamento della coda. I nodi sono invece rappresentati dalla classe **LinkedListNode**.

La struttura dati implementata ha una referenza al primo elemento della lista (*head*) e all'ultimo elemento (*tail*).

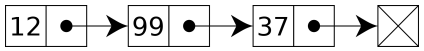

Figura 2: Esempio di lista concatenata (Fonte: Wikipedia - Linked Lists)

Il vantaggio di questa struttura dati è che inserire un nuovo valore è molto veloce, in quanto il nuovo nodo viene inserito in testa, con una complessità di $\Theta(1)$.

La rimozione di un valore invece, anche in caso di una semplice coda FIFO (in cui la priorità quindi non è data dal valore, bensì dall'ordine d'inserimento) non ha la stessa complessità, in quanto il valore inserito da più tempo si troverà in fondo alla coda (in posizione *tail*), ma per poterlo rimuovere è necessario modificare il campo *next* del nodo precedente, e quindi è necessario scorrere la lista fino a trovare l'elemento precedente a quello da rimuovere. Questo problema potrebbe essere ovviato implementando una lista doppiamente concatenata (rappresentata in Figura 3), in cui ogni nodo ha un riferimento sia al nodo precedente che a quello successivo.

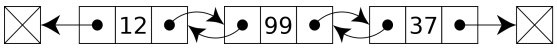

Figura 3: Lista doppiamente concatenata (Fonte: Wikipedia - Linked Lists)

Questo non è necessario nella coda con priorità considerata in questa relazione, in quanto l'elemento con priorità più alta (indicata dal valore (*value*) più alto) potrebbe trovarsi in qualsiasi posizione della lista. La ricerca del nodo con priorità massima e la sua rimozione ha quindi complessità di $\Theta(n)$, perché è sempre necessario scorrere tutti gli n elementi della lista per poter trovare il valore a priorità maggiore. Dopo averlo individuato, è necessario modificare il campo *next* del nodo precedente per farlo puntare al nodo successivo (*next*) del nodo rimosso (come rappresentato in Figura 4).

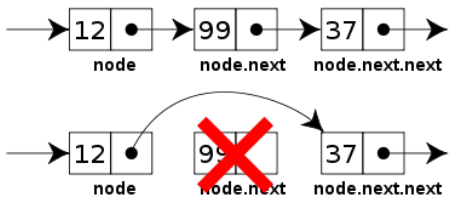

Figura 4: Rimozione di un nodo in una lista concatenata (Fonte: Wikipedia - Linked Lists)

2.3 Lista concatenata ordinata

La lista concatenata ordinata ha la stessa struttura di una lista concatenata, ma i valori sono ordinati in ordine decrescente.
Sarà quindi necessario modificare l'implementazione dell'inserimento di un nuovo nodo nella lista, scorrendo la lista fino a trovare la posizione corretta (come rappresentato in Figura 5). Il caso peggiore è quando il dato da inserire è minore di tutti gli n nodi presenti nella lista. Di conseguenza la complessità dell'inserimento è $O(n)$.

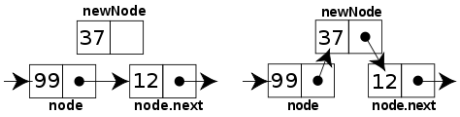

Figura 5: Inserimento in una lista concatenata ordinata (Fonte: Wikipedia - Linked Lists)

Nell'implementazione effettuata, la classe che rappresenta la lista concatenata ordinata è **SortedLinkedList** in Figura 1, che eredita direttamente dalla classe LinkedList.

I valori sono stati ordinati in modo decrescente in modo tale da avere l'elemento a priorità più alta (e quindi con valore più alto) in testa alla coda. Questo facilita la rimozione dell'elemento con valore massimo, in quanto sarà sufficiente modificare la testa della lista. Se i valori fossero ordinati in ordine decrescente, l'accesso all'elemento con priorità massima sarebbe stato semplice (elemento in posizione *tail*), ma la sua rimozione avrebbe richiesto lo scorrimento di tutta la lista, in quanto per rimuovere un nodo è necessario modificare il suo predecessore. Per mantenere la stessa complessità utilizzando un ordinamento crescente si sarebbe dovuta implementare una lista doppiamente collegata (rappresentata in Figura 3), in cui il predecessore dell'ultimo elemento è facilmente accessibile.

2.4 Max Heap

Un **Heap** è una struttura dati ad albero binario quasi completo, ossia tutte le foglie dell'albero si trovano nell'ultimo o nel penultimo livello.

L'altezza di un nodo è il numero di archi nel cammino semplice più lungo dal nodo ad una foglia. L'altezza dell'heap coincide con l'altezza della radice dell'albero ed è $\Theta(\log_2(n))$.
La struttura dati Heap è utilizzata anche negli algoritmi di ordinamento, ad esempio nell'**heapsort**.

Nel caso di una coda con priorità implementata tramite un heap, il valore del nodo inserito nell'heap indica la priorità.

Esistono due tipi di heap: **Max Heap** e **Min Heap**. Il Max Heap è un heap in cui ogni nodo ha un valore maggiore o uguale a quello dei nodi figli (come in Figura 6), mentre un Min Heap è un heap nella quale ogni nodo ha valore minore o uguale a quello dei figli.

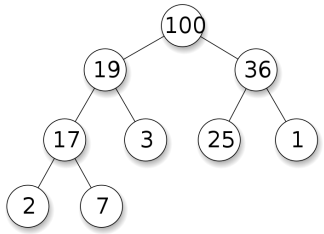

Figura 6: Esempio di un albero Max Heap (Fonte: Wikipedia - Max Heap)

Nell'implementazione della coda con priorità è stato considerato che i valori alti indicassero una priorità alta, di conseguenza la struttura dati scelta è stata un Max Heap. In questo modo l'elemento a priorità più alta si troverà nella radice dell'albero.

I nodi dell'heap possono essere memorizzati in un array. Questa struttura permette di trovare facilmente i nodi connessi a un qualsiasi nodo, dato l'indice all'interno dell'array. In particolare, considerando 0 come la prima posizione dell'array:

- Radice dell'albero: posizione 0 dell'array
- Padre del nodo i-esimo: posizione $\lfloor \frac{i-1}{2} \rfloor$
- Figlio sinistro del nodo i-esimo: posizione $2 \times i + 1$
- Figlio destro del nodo i-esimo: posizione $2 \times i + 2$

Le Figure 7 e 8 rappresentano come l'heap è effettivamente salvato in memoria. L'implementazione del Max Heap è rappresentata nel diagramma delle classi in Figura 1 come **MaxHeap**.

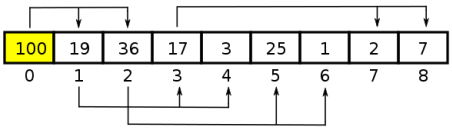

Figura 7: Rappresentazione in memoria del Max Heap in Figura 6 (Fonte: Wikipedia - Max Heap)

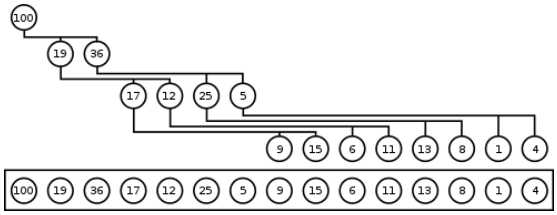

Figura 8: Confronto tra albero Max Heap e rappresentazione in memoria (Fonte: Wikipedia - Max Heap)

Oltre alle funzionalità necessarie al funzionamento della coda con priorità, sono stati implementati alcuni metodi aggiuntivi:

- `left_child(index)` (metodo statico): ritorna la posizione del figlio sinistro dell'elemento in posizione *index*
- `right_child(index)` (metodo statico): ritorna la posizione del figlio destro dell'elemento in posizione *index*
- `parent(index)` (metodo statico): ritorna la posizione del nodo padre dell'elemento in posizione *index*
- `max_heapify(index)`: serve a conservare la proprietà del max heap. Prima della chiamata a questo metodo, il nodo in posizione *index* potrebbe essere più piccolo dei suoi figli. Dopo averlo chiamato, il sottoalbero con radice *index* è un max heap
- `build_max_heap(array)`: dato un array non ordinato di valori, crea un max heap con quei valori come nodi
- `increase_key(index, value)`: incrementa il valore del nodo in posizione *index*, portandolo al valore passato come parametro (*value*) e mantenendo la struttura di max heap

2.5 Prestazioni attese

Sono state confrontate le performance delle operazioni principali delle code con priorità: l'inserimento di un nuovo valore, la ricerca del valore massimo e la sua estrazione.

2.5.1 Inserimento

Nel caso di una lista concatenata semplice l'inserimento avviene sempre in testa, quindi la complessità d'inserimento è $\Theta(1)$.

In una lista concatenata ordinata bisogna scorrere gli elementi della lista fino a trovare la posizione corretta del nuovo valore. Nel caso migliore l'inserimento avviene in testa, nel caso peggiore invece l'elemento viene inserito in fondo alla lista, dopo aver scorso tutti gli n elementi. La complessità sarà quindi $O(n)$.

Per effettuare l'inserimento di un valore in un heap si inserisce un nuovo nodo nell'ultimo livello con valore $-\infty$ in tempo costante, per poi impostarne il valore tramite `increase_key(index, value)`, che garantisce il mantenimento delle proprietà del max heap. La complessità dell'inserimento è quindi $O(\log_2(n))$.

2.5.2 Ricerca del valore massimo

In una lista concatenata semplice gli elementi non hanno un ordine, di conseguenza per trovare il nodo con valore massimo è necessario scorrere tutti i valori. La complessità è quindi $\Theta(n)$.

In una lista concatenata ordinata, dato che i valori sono ordinati in modo decrescente, l'elemento con valore massimo si troverà nella prima posizione (*head*). L'accesso a questo elemento avverrà in tempo costante (complessità $\Theta(1)$).

In un max heap l'elemento con valore massimo si trova nella radice dell'albero, in posizione 0 della lista memorizzata. La complessità della ricerca del massimo è quindi $\Theta(1)$.

2.5.3 Estrazione del valore massimo

In una lista concatenata semplice, dopo aver trovato il valore da rimuovere, questo dev'essere scollegato dalla lista come mostrato in Figura 4. Il caso peggiore è lo stesso della ricerca del valore massimo, di conseguenza la complessità è $\Theta(n)$.

Nella lista concatenata ordinata, come nel caso della ricerca, il costo della rimozione è costante ($\Theta(1)$) in quanto è necessario soltanto aggiornare il primo elemento (*head*) della lista.

Nell'heap è necessario rimuovere il valore alla radice per poter estrarre il valore massimo. Per mantenere le proprietà del max heap, l'elemento nella radice viene sostituito con l'ultimo elemento della lista memorizzata, che è una foglia, e poi viene chiamato il metodo `max_heapify(0)`, che sistema l'albero per farlo rimanere un max heap. Il costo di questa operazione è $O(\log_2(n))$, in quanto

la foglia inserita come radice scenderà l'albero di un livello alla volta finché l'albero non sarà di nuovo un max heap, e il massimo numero di livelli presenti nell'albero è $\log_2(n)$.

Struttura dati	Inserimento	Ricerca massimo	Estrazione massimo
Lista concatenata	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Lista concatenata ordinata	$O(n)$	$\Theta(1)$	$\Theta(1)$
Max Heap	$O(\log_2(n))$	$\Theta(1)$	$O(\log_2(n))$

Tabella 1: Prestazioni attese

3 Descrizione dei test effettuati

Per ogni implementazione della coda con priorità, ogni funzione è stata testata con un numero incrementale di valori, partendo da 1 fino a raggiungere i 1000 valori, con un salto di 50 valori ad ogni iterazione. Di conseguenza sono stati svolti in totale 20 test, con il numero di valori riportato nella prima colonna delle Tabelle in sezione 5.

Per ottenere dei valori più affidabili, ogni test è stato eseguito 100 volte, ed è stata fatta una media dei tempi di esecuzione. In questo modo è stata ridotta l'influenza degli altri task in esecuzione sulla macchina durante l'esecuzione dei test.

La misurazione dei tempi di esecuzione è stata effettuata tramite il metodo `timeit(stmt, number)` del modulo Python `timeit`. Il primo parametro (*stmt*) è la funzione da eseguire, mentre il secondo parametro (*number*) è il numero di volte che verrà eseguita. La funzione ritorna il tempo di esecuzione totale dei test effettuati.

I test funzionano su una copia della coda ricevuta come parametro, perché eseguono ripetutamente delle operazioni che modificano la struttura della lista. Se questo non accadesse, dopo ogni operazione la lista risulterebbe diversa rispetto alla precedente iterazione, risultando in un test errato.
Ad esempio, se in una coda con 100 elementi venisse testata 100 volte l'operazione i estrazione del massimo, al termine dei test la coda risulterebbe vuota. Creando una copia della lista prima di ogni test invece, l'estrazione del massimo avviene ogni volta su una coda con lo stesso numero di elementi.

4 Grafici dei tempi di esecuzione

Di seguito sono riportati i grafici che permettono il confronto tra le prestazioni misurate riguardo le varie implementazioni della coda. I grafici mostrano la variazione del tempo di esecuzione delle funzioni della coda con priorità, al variare del numero di elementi.

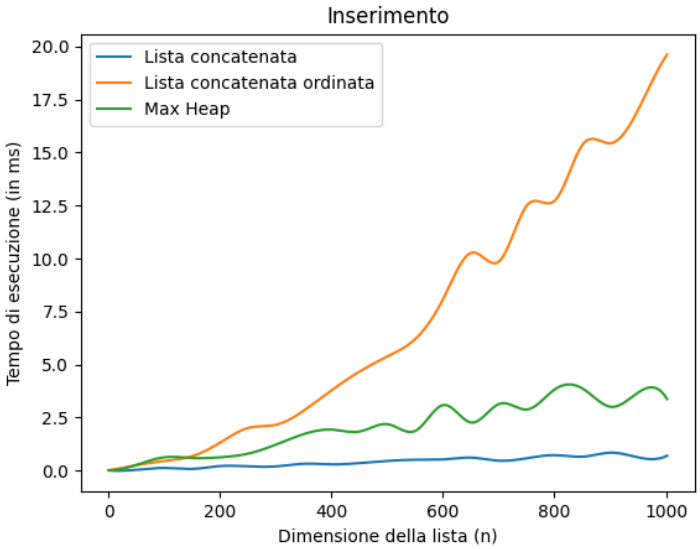

Figura 9: Tempi di esecuzione dell'inserimento di n valori nella coda

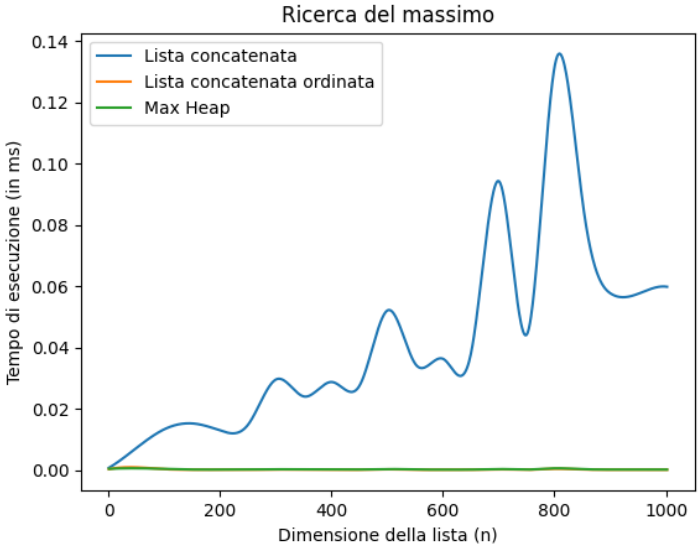

Figura 10: Tempi di esecuzione della ricerca del valore massimo al variare del numero di elementi nella coda

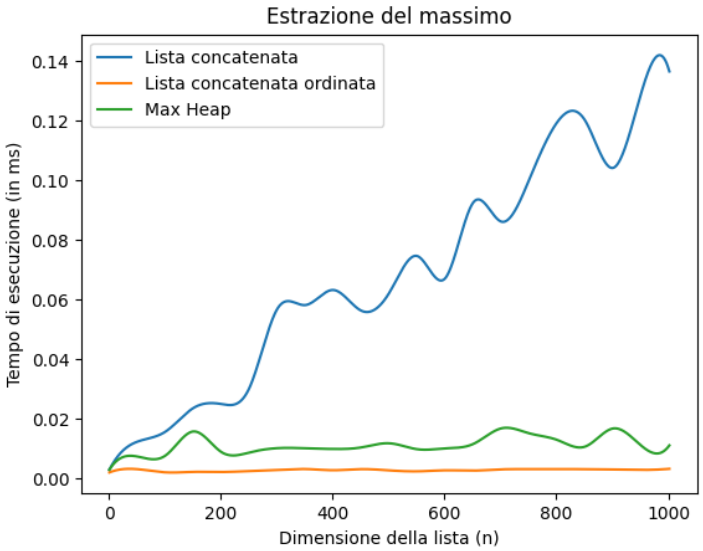

Figura 11: Tempi di esecuzione dell'estrazione del valore massimo al variare del numero di elementi nella coda

5 Tabelle dei tempi di esecuzione

Di seguito sono riportate le tabelle contenenti i tempi di esecuzione delle operazioni svolte sulla coda con priorità con la quale sono stati tracciati i grafici nella sezione 4, al variare del numero di elementi.

N° elementi	Lista concatenata	Lista concatenata ordinata	Max Heap
1	7.687e-03	7.641e-03	1.985e-02
51	3.827e-02	2.560e-01	2.721e-01
101	1.223e-01	4.543e-01	6.268e-01
151	7.799e-02	6.849e-01	5.912e-01
201	2.146e-01	1.329e+00	6.261e-01
251	1.956e-01	2.014e+00	8.001e-01
301	1.985e-01	2.165e+00	1.232e+00
351	3.215e-01	2.846e+00	1.727e+00
401	2.922e-01	3.811e+00	1.928e+00
451	3.540e-01	4.683e+00	1.847e+00
501	4.570e-01	5.395e+00	2.184e+00
551	5.093e-01	6.233e+00	1.887e+00
601	5.257e-01	8.157e+00	3.093e+00
651	6.102e-01	1.028e+01	2.262e+00
701	4.628e-01	9.885e+00	3.151e+00
751	5.881e-01	1.254e+01	2.884e+00
801	7.264e-01	1.276e+01	3.849e+00
851	6.564e-01	1.542e+01	3.847e+00
901	8.437e-01	1.544e+01	3.004e+00
951	6.107e-01	1.711e+01	3.709e+00
1001	6.992e-01	1.961e+01	3.371e+00

Figura 12: Tempi di esecuzione dell'inserimento di n valori nella coda (in ms)

N° elementi	Lista concatenata	Lista concatenata ordinata	Max Heap
1	8.160e-04	3.710e-04	4.850e-04
51	7.387e-03	9.640e-04	6.550e-04
101	1.346e-02	3.680e-04	4.710e-04
151	1.529e-02	1.930e-04	2.520e-04
201	1.297e-02	1.790e-04	2.330e-04
251	1.506e-02	1.830e-04	2.380e-04
301	2.970e-02	2.640e-04	3.400e-04
351	2.407e-02	2.040e-04	2.660e-04
401	2.881e-02	2.050e-04	2.640e-04
451	2.794e-02	1.820e-04	2.590e-04
501	5.225e-02	3.270e-04	4.190e-04
551	3.446e-02	1.880e-04	2.400e-04
601	3.625e-02	1.840e-04	2.350e-04
651	3.935e-02	1.840e-04	2.350e-04
701	9.408e-02	3.110e-04	4.140e-04
751	4.478e-02	1.850e-04	2.370e-04
801	1.330e-01	4.980e-04	6.520e-04
851	8.584e-02	2.760e-04	3.640e-04
901	5.757e-02	1.860e-04	2.430e-04
951	5.772e-02	1.830e-04	2.400e-04
1001	5.989e-02	1.830e-04	2.360e-04

Figura 13: Tempi di esecuzione della ricerca del valore massimo da una coda di n elementi (in ms)

N° elementi	Lista concatenata	Lista concatenata ordinata	Max Heap
1	2.918e-03	2.008e-03	2.941e-03
51	1.233e-02	3.106e-03	7.355e-03
101	1.567e-02	2.039e-03	7.653e-03
151	2.362e-02	2.199e-03	1.577e-02
201	2.497e-02	2.159e-03	8.861e-03
251	3.034e-02	2.463e-03	8.707e-03
301	5.692e-02	2.851e-03	1.016e-02
351	5.818e-02	3.155e-03	1.012e-02
401	6.323e-02	2.752e-03	9.882e-03
451	5.628e-02	3.119e-03	1.041e-02
501	6.234e-02	2.695e-03	1.179e-02
551	7.462e-02	2.375e-03	9.794e-03
601	6.723e-02	2.726e-03	1.009e-02
651	9.264e-02	2.622e-03	1.158e-02
701	8.617e-02	3.041e-03	1.681e-02
751	1.004e-01	3.123e-03	1.513e-02
801	1.196e-01	3.137e-03	1.290e-02
851	1.200e-01	3.106e-03	1.071e-02
901	1.042e-01	2.990e-03	1.678e-02
951	1.295e-01	2.858e-03	1.097e-02
1001	1.366e-01	3.229e-03	1.109e-02

Figura 14: Tempi di esecuzione dell'estrazione del valore massimo da una coda di n elementi (in ms)

6 Osservazioni sui risultati

6.1 Inserimento

La tabella in Figura 12 mostra i valori dei tempi di esecuzione dell'inserimento nelle diverse implementazioni della coda con priorità.
Tramite questi valori è stato creato il grafico in Figura 9, che mostra visivamente la complessità dell'inserimento.

Come previsto, l'inserimento in una lista concatenata semplice ha costo costante, e i tempi di esecuzione sono molto inferiori rispetto alle altre due implementazioni.

Con la lista concatenata ordinata invece, l'inserimento è un'operazione molto costosa. Dalla linea arancione in Figura 9 è possibile notare che i tempi di esecuzione aumentano linearmente all'aumentare del numero di valori nella lista.

Il costo dell'inserimento in un heap cresce in modo nettamente inferiore rispetto a una lista concatenata ordinata, infatti la complessità dell'inserimento nell'heap è $O(\log_2(n))$, mentre nella lista concatenata ordinata è $O(n)$. Il costo rimane comunque superiore rispetto all'inserimento nella lista concatenata semplice, in cui la complessità è $\Theta(1)$.

6.2 Ricerca del valore massimo

La tabella in Figura 13 mostra i valori dei tempi di esecuzione della ricerca del valore massimo.
Tramite questi valori è stato creato il grafico in Figura 10, che mostra visivamente la complessità della ricerca.

A differenza dell'inserimento, con una lista concatenata semplice la ricerca del massimo è un'operazione molto dispendiosa. Dal grafico infatti è possibile delineare una crescita lineare dei tempi di esecuzione, come previsto dalla sezione teorica.

Con la lista concatenata ordinata e con l'heap, data la loro struttura, la ricerca del valore massimo avviene con un numero costante di operazioni (quindi con complessità $\Theta(1)$). L'andamento del loro grafico è infatti costante, con valori dei tempi di esecuzione (riportati nella tabella 13) di due ordini di grandezza inferiori rispetto a quelli della lista concatenata semplice.

6.3 Estrazione del valore massimo

La tabella in Figura 14 mostra i valori dei tempi di esecuzione dell'estrazione del valore massimo.
Tramite questi valori è stato creato il grafico in Figura 11, che mostra visivamente la complessità dell'estrazione.

Nella lista concatenata, come nel caso della ricerca del valore massimo, i tempi di esecuzione crescono linearmente al crescere dei valori nella coda.

Il costo dell'estrazione del massimo rimane lo stesso della ricerca nel caso della lista concatenata ordinata, mentre l'heap ha un costo maggiore in quanto dopo aver rimosso l'elemento nella radice si devono effettuare delle operazioni per mantenere le proprietà di max heap, che hanno un costo logaritmico.

6.4 Conclusioni

L'analisi della complessità degli algoritmi effettuata a livello teorico è stata confermata dai test e dalle misurazioni svolte sperimentalmente.

Con una lista concatenata l'inserimento è molto rapido, ma è lenta la ricerca ed estrazione del valore massimo.

In lista concatenata ordinata l'inserimento è lento inserimento, ma l'estrazione del valore massimo è rapida.

Un Max Heap offre un compromesso tra le strutture dati considerate, in quanto sia l'inserimento che l'estrazione del massimo hanno una complessità logaritmica, e la ricerca del valore massimo avviene in tempo costante.

Riferimenti bibliografici

[1] Wikipedia - Linked List (URL)
Fonte delle Figure 2, 3, 4, 5

[2] PrepInsta - Sorted Linked Lists (URL)

[3] Wikipedia - Heap (URL)
Fonte delle Figure 6, 7, 8