UNIVERSITÀ
DEGLI STUDI
FIRENZE

Università degli studi di Firenze

LABORATORIO DI ALGORITMI E STRUTTURE DATI

Autore: Ivan Necerini
Numero di matricola: 7049380
Corso principale: Algoritmi e strutture dati
Docente: Simone Marinai

May 2023

Indice

1 Introduzione

Lo scopo della presente relazione è confrontare diverse implementazioni della coda con priorità. Sono state realizzate tre diverse strutture dati per implementare la coda con priorità: il **max heap**, la **lista concatenata** e la **lista concatenata ordinata**.
Verranno analizzate e confrontate le prestazioni delle operazioni di inserimento, ricerca del massimo ed estrazione del massimo, che sono le operazioni più significative, su una coda con priorità implementata utilizzando le tre diverse strutture dati menzionate in precedenza.

1.1 Specifiche della piattaforma di test

Il codice per eseguire gli esperimenti è stato scritto in Python (interprete v.3.10.2, ultima versione a oggi), utlizzando l'ambiente di sviluppo "Microsoft Visual Studio Code" v.1.78.2.
Il computer su cui è stato scritto il codice ha le seguenti specifiche:

- Processore 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz
- RAM installata 16,0 GB (15,8 GB utilizzabile)
- Sistema operativo a 64 bit, processore basato su x64
- Windows 11 Home Versione 22H2

La presente relazione è stata scritta in LaTeXtramite l'editor online Overleaf.
Il diagramma delle classi in Figura 2 è stato disegnato tramite il software Lucidchart.

1.2 Librerie utilizzate

Segue una lista delle librerie principali utilizzati nella scrittura del codice python:

- `numpy`: per generare array con valori casuali

  (funzioni usate: numpy.random.randint())
- `os`: per creare le cartelle dove salvare le immagini di grafici e tabelle

  (funzioni usate: os.path.exists() e os.makedirs())
- `sys`: utilizzata per aumentare il limite di ricorsione massimo, necessario per evitare che venga generato un errore di ricorsione

  (funzioni usate: sys.setrecursionlimit())
- `anyTree`: offre strumenti per lavorare con alberi gerarchici, ovvero collezioni di nodi interconnessi da archi. La funzione Node() di anyTree crea un nuovo nodo con il nome specificato e l'aggiunge all'albero. La funzione RenderTree() restituisce una rappresentazione in stringa dell'albero con i nodi indentati in modo da riflettere la loro gerarchia.

  (funzioni usate: anyTree.Node() e anyTree.RenderTree())

- `copy`: utilizzata per fare copie profonde di oggetti

  (funzioni usate: copy.deepcopy())

- `functools`: utilizzata per lavorare con funzioni di ordine superiore e oggetti chiamabili. La funzione 'partial' si usa poi per creare nuove funzioni parzialmente applicate a partire da una funzione esistente

  (funzioni usate: functools.partial())

- `timeit`: utilizzata per misurare i tempi di esecuzione degli algoritmi

  (funzioni usate: timeit.timeit(stmt, number))

- `matplotlib`: utilizzata per generare grafici e tabelle in modo da vedere effettivamente le prestazioni degli algoritmi

  (funzioni usate: matplotlib.pyplot.plot() e matplotlib.pyplot.table())

2 Cenni teorici

Una **coda con priorità** è una struttura dati utilizzata per gestire un insieme dinamico di elementi denominato "S". Ogni elemento in questa coda è associato a un valore chiamato "chiave", che determina la sua priorità. Nei nostri esperimenti, valori più alti di chiave indicano una priorità più elevata.
Le operazioni fondamentali per il corretto funzionamento e utilizzo della coda con priorità includono l'inserimento di elementi, la ricerca del valore massimo e la sua eventuale estrazione. In questa relazione, analizzeremo e confronteremo queste operazioni nelle diverse implementazioni della coda con priorità.

2.1 Schema delle classi, organizzazione del codice e delle strutture dati

Il diagramma delle classi è rappresentato in figura 2, al fine di spiegare al meglio l'organizzazione del codice Python.

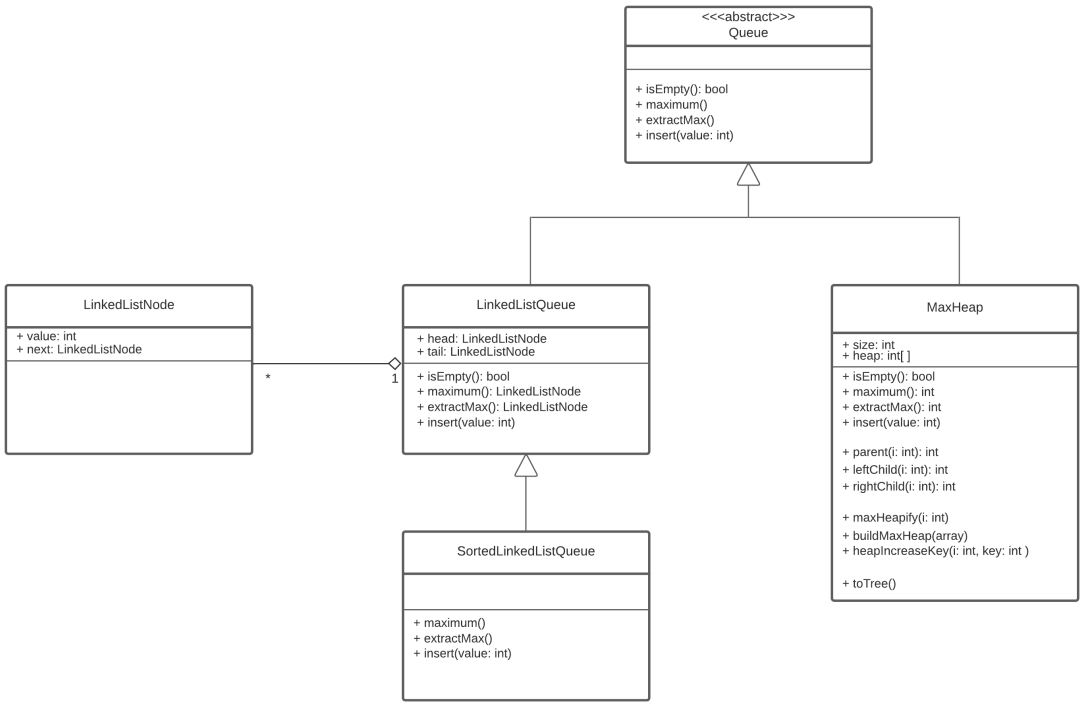

Figura 2: Diagramma delle classi

La relazione ha lo scopo di condurre test su tre operazioni di una coda con priorità, implementata utilizzando diverse strutture dati. Per raggiungere questo obiettivo, è stata creata una classe astratta chiamata "Queue". La classe "Queue" contiene i prototipi dei metodi fondamentali per il corretto funzionamento di una coda con priorità. Ciò impone alle classi derivate (che

sono concrete) di implementare effettivamente tali metodi, al fine di rendere la coda con priorità utilizzabile con le diverse strutture dati implementate.

2.1.1 Max heap

Un heap binario (classe *MaxHeap* in figura 2 è una struttura dati composta da un array che può essere considerato come un albero binario quasi completo (per questo mtoivo la sua altezza è di $\Theta(\log_2(n))$) come mostrato nella figura 3, dove il numero all'interno del cerchio è il valore registrato dentro quel nodo e il numero sopra è l'indice del corrsipondente array. Ogni nodo dell'albero corrisponde a un elemento dell'array che memorizza il valore del nodo. Tutti i livelli dell'albero sono completamente riempiti, tranne l'ultimo che potrebbe non essere completamente riempito dalla sinistra. Un array H che rappresenta un heap è un oggetto con due attributi: lunghezza[H] che indica il numero di elementi nell'array e size[H] che indica il numero di elementi dell'heap registrati nell'array H. In altre parole, anche se H[1...lunghezza[H]] contiene numeri validi, nessun elemento dopo H[size[H]], dove size[H] $\leq$ lunghezza[H], è un elemento dell'heap.

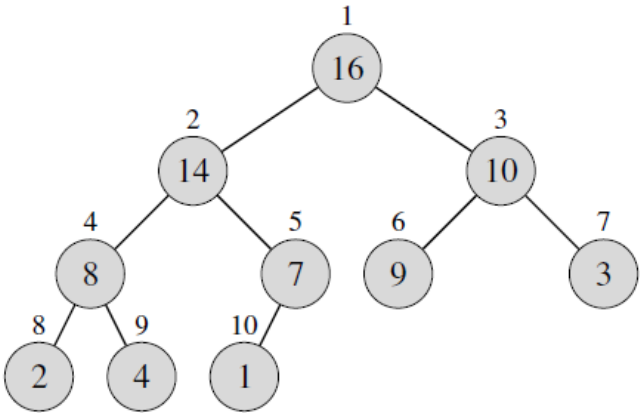

Figura 3: MaxHeap visto come un albero binario

[1]

Ci sono due tipi di heap binari: maxHeap e minHeap. In entrambi i valori nei nodi soddisfano una proprietà dell'heap, le cui caratteristiche dipendono dal tipo di heap. In un maxHeap si ha che H[PARENT(i)] $\geq$ H[i], ovvero il valore di un nodo è al massimo il valore del suo padre, per ogni nodo. Quindi, l'elemento più grande di un maxHeap è memorizzato nella radice (posizione 0 array). Nell'implementazione del codice si è considerato un maxHeap come struttura dati.
Nella figura 4 possiamo vedere un maxHeap visto invece come array. Sopra e sotto l'array ci sono delle linee che rappresentano le relazioni padre e figlio, i padri sono sempre alla sinistra dei loro figli.

[1]Fonte dell'immagine: *Introduzione agli algoritmi e strutture dati*, pagina 128, Cormen, Leiserson, Rivest e Stein, *McGrawHill*, 2010.

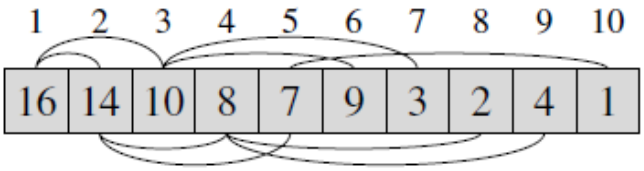

Figura 4: MaxHeap visto come un array

[2]

Grazie alla rappresentazione di un heap visto come un array, è possibile risalire facilmente ai nodi connessi a un qualisasi altro nodo, nello specifico:

- 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 $2 \times i + 2$

Nel codice, oltre ai tre metodi statici (parent(i), leftChild(i) e rightChild(i)) usati per ritornare le posizioni di nodi particolari visti poco sopra, sono stati scritti altri metodi utili per il funzionamento delle operazioni da testare sulla coda con priorità:

- `maxHeapify(i)`: serve a mantenere la proprietà di Max heap. Quando viene chiamato assume che gli alberi binari con radice in leftChild(i) e rightChild(i) siano Max heap ma che H[i] possa essere più piccolo dei suoi figli, violando la proprietà fondamentale di Max heap. Il metodo maxHeapify(i) fa scendere nell'albero il valore H[i] in modo che il sottoalbero con radice in i diventi un Max heap
- `buildMaxHeap(array)`: riceve come argomento un array e inizializza l'heap con tale array, impostando la dimensione dell'heap uguale alla lunghezza dell'array. Successivamente, attraverso un ciclo che parte dalla metà dell'heap e arriva a 0, viene chiamata la funzione maxHeapify su ciascun nodo dell'heap. La chiamata a maxHeapify viene effettuata solo sui nodi che non sono foglie dell'heap, in quanto i nodi foglia sono considerati come max-heap (poiché non hanno figli). Quindi, la funzione buildMaxHeap crea un max-heap a partire dall'array fornito, garantendo che tutti i nodi soddisfino la proprietà del max-heap.
- `heapIncreaseKey(i, key)`: La funzione heapIncreaseKey è utilizzata per aumentare il valore di un nodo specifico nell'heap, mantenendo la proprietà del max-heap. Questa funzione viene comunemente chiamata quando si desidera aumentare la priorità di un elemento nell'heap. Questo metodo è *fondamentale in quanto viene chiamato nell'inserimento di un valore in una coda di priorità implementata con Max heap*

[2]Fonte dell'immagine: *Introduzione agli algoritmi e strutture dati*, pagina 128, Cormen, Leiserson, Rivest e Stein, *McGrawHill*, 2010.

2.1.2 Rappresentazione Max heap come albero a fini di debug

Nel codice, ai fini di debug di funzionamento del file *maxHeap.py*, è stata usata la libreria **anyTree**.
La libreria anytree per creare una rappresentazione ad albero dei nodi presenti nell'heap. Questa libreria fornisce un modo conveniente per gestire alberi gerarchici in Python. In particolare, il metodo toTree della classe MaxHeap utilizza la libreria anytree per convertire l'array dell'heap in una struttura ad albero.
Il metodo toTree crea un oggetto albero utilizzando la classe Node fornita dalla libreria anytree. Inizialmente, viene creato un nodo radice con il valore del primo elemento dell'heap (self.heap[0]). Quindi, viene utilizzata una coda per esplorare l'heap e creare i nodi figli corrispondenti ai nodi sinistri e destri di ciascun nodo nell'heap. I nodi figli vengono collegati al nodo corrente utilizzando il concetto di genitore/figlio nella libreria anytree.
Infine, il metodo restituisce il nodo radice dell'albero creato.
L'utilizzo della libreria anytree e del metodo toTree consente di visualizzare graficamente la struttura ad albero dell'heap. Nell'esempio di codice fornito, dopo aver eseguito alcune operazioni sull'heap (come l'inserimento e l'estrazione del valore massimo), viene chiamato il metodo toTree per ottenere la rappresentazione ad albero dell'heap corrente. Questo albero viene quindi visualizzato utilizzando il modulo RenderTree della libreria anytree.
L'utilizzo di anytree e del metodo toTree consente di visualizzare l'heap in una forma più comprensibile, facilitando l'ispezione e il debugging della struttura dell'heap. Inoltre, la rappresentazione ad albero può essere utile per visualizzare la gerarchia dei valori presenti nell'heap.

2.1.3 Lista concatenata

Una lista concatenata è una struttura dati utilizzata per rappresentare una sequenza di elementi collegati tra loro mediante puntatori. A differenza di un array, dove gli elementi sono memorizzati in posizioni contigue di memoria, in una lista concatenata ogni elemento è un nodo che contiene il valore dell'elemento stesso e un puntatore al successivo nodo nella lista. Il puntatore "successivo" punta al nodo successivo nella lista. L'ultimo nodo della lista ha un puntatore "successivo" nullo o vuoto, indicando la fine della lista, come si vede in figura 5.

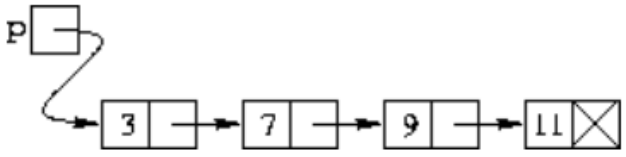

Figura 5: Lista singolarmente concatenata

3

[3]Fonte dell'immagine: *https://www.science.unitn.it/ brunato/labpro1/lista.html*

Nel codice, dove viene implementata una coda con priorità tramite lista concatenata, il dato è un valore che scandisce la priorità del nodo. La lista concatenata (classe *LinkedListQueue* in figura 2) è stata realizzata implementando tutti i metodi che non erano stati scritti nella classe astratta *Queue* e ogni nodo è rappresentato dalla classe *LinkedListNode* in figura 2.
Questa struttura dati ha diversi vantaggi e svantaggi, che si qualificano in base all'efficienza che hanno nelle varie operazioni standard:

- `Inserimento di un valore in testa`: operazione velocissima con una complessità di $\Theta(1)$.

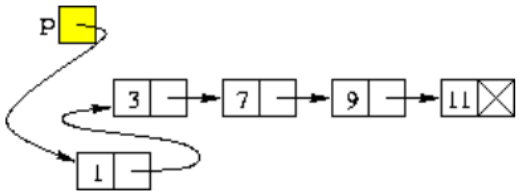

Figura 6: Inserimento in testa a una lista concatenata

[4]

- `Rimozione di un valore`: in un caso generico di rimozione di un elemento con una certa chiave da una lista concatenata non ordinata (come si vede in figura 7), si devono seguire i seguenti passaggi:

  1. Iniziare dal primo elemento della lista e confrontare la chiave dell'elemento con la chiave che deve essere rimossa.
  2. Se la chiave corrisponde, si può rimuovere l'elemento dalla lista aggiornando però i puntatori degli elementi precedenti e successivi all'elemento che vuoi rimuovere. Ad esempio, se l'elemento da rimuovere è il secondo in una lista, si deve impostare il puntatore del primo elemento al terzo elemento.
  3. Se la chiave non corrisponde, si passa all'elemento successivo nella lista e si ripete il passaggio 2.
  4. Continuare a seguire i passaggi 2 e 3 finché non si è controllato tutti gli elementi nella lista o fino a quando non si è trovato e rimosso l'elemento desiderato.

  Nella presente relazione, è necessario individuare ed estrarre il valore massimo all'interno di una coda con priorità implementata tramite una lista concatenata. Il valore massimo corrisponde all'elemento con il *value* più elevato, ovvero con la priorità maggiore rispetto a tutti gli altri nodi. Poiché il valore massimo potrebbe trovarsi in qualsiasi posizione della lista, sia la sua ricerca che la sua rimozione richiedono un costo di Θ(n). Ciò è dovuto al fatto che è necessario scorrere tutti e "n" gli elementi della lista. Inoltre, dopo la rimozione,

[4]Fonte dell'immagine: *https://www.science.unitn.it/ brunato/labpro1/lista.html*

sarà indispensabile aggiornare il campo *next* del nodo precedente, assegnandogli il valore del campo *next* del nodo appena rimosso.

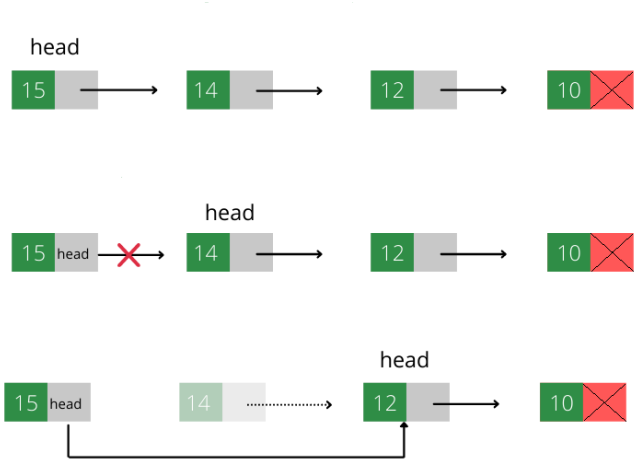

Figura 7: Cancellazione del secondo elemento da una lista concatenata

[5]

2.1.4 Lista concatenata ordinata

Una lista concatenata ordinata (nel codice è la classe *SortedLinkedList* che eredita da *LinkedList* come si vede dala figura 2) è analoga alla lista concatenata descritta precedentemente con la sola differenza che i valori sono ordinati in senso decrescente (nel caso di questa relazione e quindi di una scelta implementativa).
Analizziamo dunque le differenze implementative delle operazioni di inserimento e cancellazione:

- `Inserimento di un valore`: non sarà sufficiente inserire il nuovo nodo a inizio lista, bensì sarà necessario scorrere la lista finchè non si trova la posizione corretta in cui andrà inserito. Il caso peggiore è quando l'elemento da inserire è più piccolo di tutti gli altri nodi della lista e duqnue si dovrà posizionare in fondo ad essa (vedi figura 8). Il costo dunque è di $O(n)$.

[5]Fonte dell'immagine: *https://www.geeksforgeeks.org/deletion-in-linked-list/*

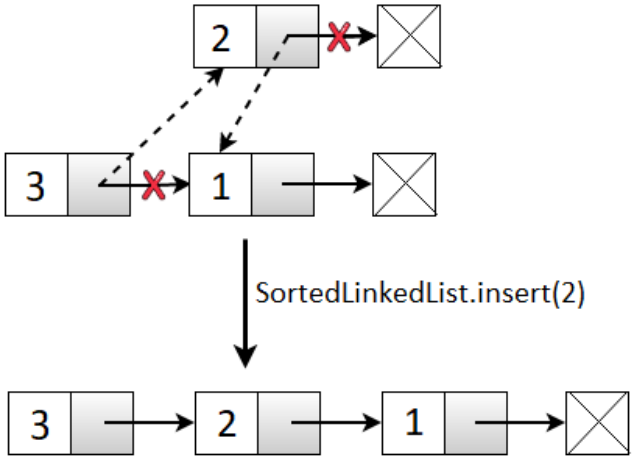

Figura 8: Inserimento di un elemento in una lista ordinata

6

- `Rimozione del valore massimo`: in questo caso, l'operazione è molto agevolata dal fatto che la lista è ordinata in senseo decrescente. Per questo motivo, basterà modificare la testa della lista per rimuovere l'elemento massimo. Quest'operazione avrà dunque una complessità pari a $\Theta(1)$. Vedi figura 9

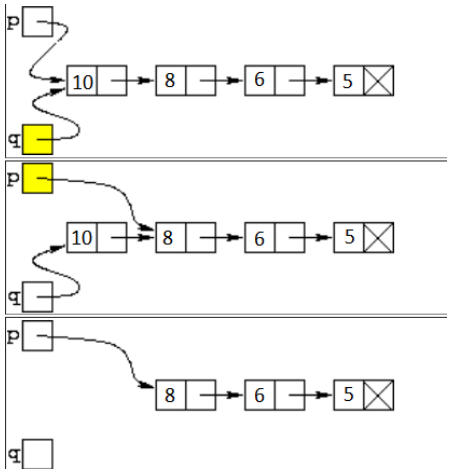

Figura 9: Cancellazione del primo elemento da una lista ordinata

7

[6]Fonte dell'immagine: *https://www.techiedelight.com/sorted-insert-in-linked-list/*
[7]Fonte dell'immagine: *https://www.science.unitn.it/ brunato/labpro1/lista.html*

2.2 Prestazioni teoriche volute

Come accennato in precedenza, questa relazione confronta le prestazioni di tre operazioni su una coda di priorità, che sono state implementate utilizzando tre diverse strutture dati. Le operazioni prese in considerazione sono l'inserimento di nuovi valori, la ricerca del valore massimo e l'estrazione di tale valore.

2.2.1 Inserimento dei valori

1. Per inserire un valore in un max heap, viene aggiunto un nuovo nodo nell'ultimo livello con un valore di $-\infty$, questo avviene in tempo costante. Successivamente, tramite l'utilizzo di *increaseKey(i, key)*, viene impostato il valore del nodo, garantendo il mantenimento delle proprietà del max heap. La complessità dell'inserimento è quindi di $O(\log_2(n))$.

2. In una lista concatenata non ordinata, l'inserimento avviene sempre in testa, rendendo la complessità dell'inserimento $\Theta(1)$.

3. In una lista concatenata ordinata, è necessario scorrere gli elementi della lista fino a trovare la posizione corretta del nuovo valore. Nel caso migliore, l'inserimento avviene in testa, mentre nel caso peggiore l'elemento viene inserito alla fine della lista dopo aver attraversato tutti gli n elementi. Di conseguenza, la complessità dell'inserimento sarà $O(n)$.

2.2.2 Ricerca del valore massimo

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

2. In una lista concatenata non ordinata, gli elementi non seguono un ordine specifico. Quindi, per trovare il nodo con il valore massimo è necessario scorrere tutti i valori. La complessità di questa operazione è quindi $\Theta(n)$.

3. In una lista concatenata ordinata, poiché i valori sono ordinati in modo decrescente, l'elemento con il valore massimo si troverà nella prima posizione, ovvero all'inizio della lista (i.e *head*). L'accesso a questo elemento avverrà in tempo costante, con una complessità di $\Theta(1)$.

2.2.3 Estrazione del valore massimo

1. Nel max heap, per estrarre il valore massimo, è necessario rimuovere l'elemento inziale. Per garantire che il max heap mantenga le sue proprietà, l'elemento iniziale viene sostituito con l'ultimo elemento della lista memorizzata (foglia). Successivamente, viene chiamato il metodo *maxHeapify(0)*, che riorganizza l'albero in modo che rimanga un max heap. Poiché la foglia inserita come radice scenderà nell'albero di un livello alla volta fino a quando l'albero non sarà nuovamente un max heap, il costo di questa operazione è $O(\log_2(n))$ ($\log_2(n)$ corrisponde al numero massimo di livello presenti nell'albero).

2. In una lista concatenata non ordinata, una volta individuato il valore da rimuovere, questo viene scollegato dalla lista come mostrato nella figura 7. Il caso peggiore ha complessità di questa operazione è $\Theta(n)$ (come la ricerca del massimo).

3. In una lista concatenata ordinata il costo della rimozione è costante ($\Theta(1)$) (uguale alla ricerca) in quanto è sufficiente aggiornare il primo elemento (ovvero *head*) della lista.

2.2.4 Tabella riassuntiva

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

Tabella 1: Prestazioni attese delle varie operazioni

3 Descrizione degli esperimenti condotti

Per ogni implementazione della coda con priorità, si sono condotti test su ogni funzione (inserimento dei valori, ricerca ed estrazione del massimo) utilizzando un incremento graduale di valori (n), da 1 fino a 600, con uno step di 30 unità ad ogni iterazione.
In totale sono stati effettuati dunque 20 test (corrispondono alla prima colonna delle tabelle in sezione 4).
Al fine di ottenere risultati più affidabili, ciascun test è stato eseguito 100 volte e i tempi di esecuzione sono stati mediati. In tal modo si è ridotta l'influenza di altri processi in esecuzione sulla macchina durante i test.
Per misurare i tempi di esecuzione, è stato utilizzato il metodo *timeit(stmt, number)* del modulo Python *timeit*. Il primo parametro (*stmt*) corrisponde alla funzione da eseguire, mentre il secondo parametro (*number*) indica il numero di iterazioni da fare. Questo metodo restituisce il tempo totale impiegato per l'esecuzione dei test.

3.1 Operazioni effettuate su una copia della coda con priorità

Nota bene: I test vengono eseguiti su una copia della coda fornita come parametro. Ciò è necessario poiché tali test coinvolgono ripetute operazioni che modificano la struttura della lista. Senza l'uso di una copia, ogni operazione influenzerebbe la lista rispetto all'iterazione precedente, portando a risultati di test errati.
Per questo motivo viene utilizzata la libreria `copy` e la sua funzione *copy.deepcopy()*, come descritto in sezione 1.2.

4 Tabelle dei tempi di esecuzione

5 Grafici dei tempi di esecuzione

6 Analisi e osservazioni sui risultati finali

6.1 Inserimento dei valori

6.2 Ricerca del valore massimo

6.3 Estrazione del valore massimo

6.4 Conclusioni