UNIVERSITÀ DEGLI STUDI FIRENZE

UNIVERSITÀ DEGLI STUDI DI FIRENZE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Relazione di Laboratorio di Algoritmi e Strutture Dati

Autore:
Alberto Pizzi

N° Matricola:
7073782

Corso principale:
Algoritmi e Strutture Dati

Docente corso:
Simone Marinai

Indice

1 Introduzione

1.1 Descrizione problema

Nel primo esercizio viene chiesto di confrontare varie implementazioni di statistiche d'ordine dinamiche, attraverso:

- un **heap**
- una **lista concatenata**
- una **lista concatenata ordinata**

Quindi andremo a confrontare il comportamento di alcuni algoritmi di statistiche d'ordine dinamiche per ognuna delle tre strutture dati. Per fare ciò andremo ad implementare le liste creandole da zero (tramite puntatori ai nodi), senza l'ausilio di librerie esterne. Per quanto riguarda l'heap, per semplicità, useremo le liste di Python, dato che non vengono date limitazioni implementative dal testo dell'esercizio.

1.2 Introduzione alla struttura generale dei test

Eseguiremo i test degli algoritmi:

- **ricerca del massimo**
- **ricerca del minimo**
- **OS-select**: restituisce il k-esimo elemento più piccolo
- **OS-rank**: restituisce il rango, ovvero la posizione di un valore in un attraversamento in ordine (crescente), dato un valore

Ognuno di questi algoritmi verrà testato su tutte le implementazioni delle strutture dati elencate precedentemente. I test verranno eseguiti per **più volte**, facendo una media, in modo tale da avere un'affidabilità maggiore nei dati raccolti. Ogni algoritmo di test (max, min, ecc...) verrà testato con 3 tipologie di input differenti, ossia:

- valori **random**
- valori **crescenti**
- valori **decrescenti**

1.3 Caratteristiche hardware

Tutti i test vengono eseguiti su una macchina con le seguenti caratteristiche

- **Sistema operativo**: Microsoft Windows 11 Pro
- **Tipo di sistema**: x64
- **CPU**: AMD Ryzen 5 3600 3.59 Ghz 6 core
- **GPU**: NVIDIA Geforce RTX 2060 SUPER
- **RAM**: 32 GB DDR4 2666Mhz
- **Memoria di archiviazione primaria**: SSD Crucial 1TB P5 CT1000P5SSD8 PCIe M.2 Nvme

Il codice è sviluppato in *Python versione 3.12* eseguito su IDE: *Pycharm 2023.3.3 Professional Edition*. La relazione in linguaggio Latex è scritta sulla piattaforma online *Overlaf*.

2 Teoria

In questo capitolo faremo dei cenni teorici agli algoritmi e strutture date utilizzate nell'esercizio, soffermandoci anche sulle prestazioni attese degli algoritmi.

2.1 Algoritmi

Adesso facciamo facciamo dei brevi cenni sulla teoria finalizzata alla comprensione degli esperimenti e riguardante gli algoritmi utilizzati per i test. Focalizzandoci su quelli finalizzati al testo dell'esercizio, ovvero le statistiche d'ordine dinamiche.

Innanzitutto per i-esima **statistica d'ordine** di un insieme di n elementi è l'i-esimo elemento più **piccolo**. Di conseguenza il minimo di un insieme è la prima statistica d'ordine e il massimo è la n-esima. Quindi estendendo il concetto ad insiemi dinamici, tipici dell'informatica, ovvero insiemi in continuo mutamento a causa di aggiunte, rimozioni o altre modifiche sui dati nel tempo, in risposta ad un processo di evoluzione.

2.1.1 Minimo

Il minimo di un insieme è l'elemento più piccolo. Quindi in un insieme **non ordinato** sarà necessario cercarlo per forza bruta ispezionando ogni valore e stabilendo il minimo temporaneo man mano che scorriamo gli elementi. Questo è necessario perché non sappiamo a priori dove si trovi.
In un insieme ordinato in modo **crescente** il minimo è il primo elemento. Invece in un insieme ordinato in modo **decrescente** il minimo è l'ultimo elemento. La modalità di accesso e il costo variano a seconda della struttura dati utilizzata.

2.1.2 Massimo

Il massimo di un insieme è l'elemento più grande. Quindi in un insieme **non ordinato** sarà necessario cercarlo per forza bruta, come per il minimo.
In un insieme ordinato in modo **crescente** il massimo è l'ultimo elemento. Invece in un insieme ordinato in modo **decrescente** massimo è il primo elemento. La modalità di accesso e il costo variano a seconda della struttura dati utilizzata.

2.1.3 OS-select

L'OS-select ricerca un elemento con un dato *rango*, ovvero la posizione che occupa nella sequenza ordinata degli elementi dell'insieme. Il costo varia a seconda della struttura dati utilizzata. Restituisce il valore (o il suo puntatore al nodo) con tale rango.

2.1.4 OS-rank

L'OS-rank restituisce il *rango* di un valore, dato il valore. Il costo varia a seconda della struttura dati utilizzata.

OS-select e OS-rank sono complementari, infatti se ho una lista ordinata di valori, l'OS-select è l'inverso dell'OS-rank e viceversa.

$$\text{i = OS-rank(OS-select(i))}$$
$$\text{key = OS-select(OS-rank(key))}$$

Quindi facendo riferimento ad un valore a , se lo diamo in input all'OS-rank ci ritornerà il suo rango. Se a sua volta diamo in input il rango fornito in output dall'OS-rank all'OS-select, il suo output sarà il valore iniziale a.

2.2 Strutture dati

Adesso facciamo facciamo dei brevi cenni sulla teoria finalizzata alla comprensione degli esperimenti e riguardante le strutture dati utilizzate per i test.

La seguente tabella (1) riassume i costi previsti nei casi peggiori (e attesi) di tutte le strutture dati per ogni algoritmo testato. La analizzeremo nei paragrafi successivi.

	Massimo	Minimo	OS-select	OS-rank
Min-heap	$O(n)$	$O(1)$	$O(n \lg n)$	$O(n \lg n)$
Max-heap	$O(1)$	$O(n)$	//	//
Lista concatenata	$O(n)$	$O(n)$	$O(n^2)$	$O(n^2)$
Lista concatenata ordinata	$O(n)$	$O(1)$	$O(n)$	$O(n)$

Tabella 1: Tabella con i costi (casi peggiori) degli algoritmi di test, relativi alle implementazioni delle strutture dati

2.2.1 Heap

L'heap è un albero binario quasi completo. Può essere memorizzato tramite un array A con:

- Radice dell'albero: $A[1]$
- Padre di $A[1]$: $A[\lfloor i/2 \rfloor]$
- Figlio sinistro di $A[i]$: $A[2i]$
- Figlio destro di $A[i]$: $A[2i+1]$

Useremo questi indici anche nell'implementazione Python. L'heap si divide principalmente in due tipi con proprietà simmetriche:

- **Max-heap**: $A[Parent(i)] \geq A[i]$, quindi ha il massimo nella radice
- **Min-heap**: $A[Parent(i)] \leq A[i]$, quindi ha il minimo nella radice

Quindi facendo riferimento al nodo genitore, stabiliamo gli altri di conseguenza a seconda del tipo di heap.

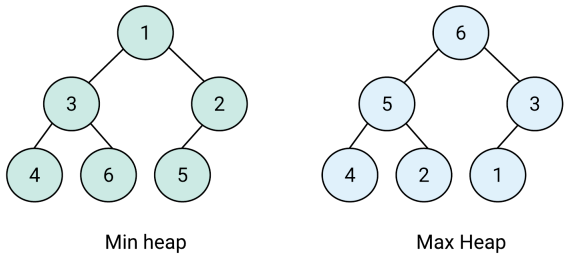


Figura 1: Esempio che rappresenta le differenze tra *max-heap* e *min-heap* con i soliti valori.

Come vediamo dall'immagine 1, viene data priorità al valore più piccolo (nel *min-heap*) o al più grande (nel *max-heap*) a seconda del tipo. Infatti l'heap è molto utile per le code con priorità.

Quindi in un *min-heap* la ricerca del **minimo** ha costo $\mathbf{O(1)}$, ugualmente per la ricerca del **massimo** in un *max-heap*.

Nel caso in cui volessi cercare il massimo in un *min-heap*, è possibile farlo nonostante sia **strutturalmente inefficiente** perché potremo implementare direttamente un *max-heap*. Infatti gli elementi del *min-heap* da $\lfloor n/2 \rfloor + 1$ fino ad n, ovvero le foglie, contengono il massimo, quindi possiamo effettuare una ricerca del massimo su metà dei nodi totali con un costo **lineare**. Vale lo stesso per la ricerca del minimo in un *max-heap*.

Dalla figura 2 vediamo, come detto precedentemente, che nella seconda metà dell'array sono presenti le foglie dell'heap. Quindi per ogni elemento possiamo trovare figli e genitori grazie all'indicizzazione esposta al paragrafo 2.2.1.

Invece per quanto riguarda l'implementazione di un **OS-select** in un *min-heap*, siccome dobbiamo trovare il k-esimo elemento più piccolo, possiamo effettuare k estrazioni del minimo. Un'estrazione del minimo ha costo $O(\lg n)$,dato dalla chiamata ad *Heapify* al suo interno, quindi per k estrazioni del minimo avremo un costo di $O(k \lg n)$. Nel peggiore dei casi se $k = n$, avremo un costo di $\boldsymbol{O(n \lg n)}$.

Invece per l'**OS-rank**, sempre implementato con un *min-heap*, dovendo restituire il rango di un dato valore in input, possiamo estrarre il minimo dalla radice affinché il valore cercato non compare

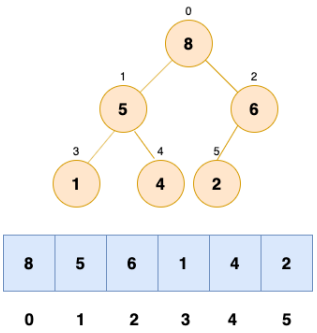


Figura 2: Esempio di memorizzazione di *max-heap* in un array.

nella radice. Quindi, come per l'OS-select, l'estrazione del minimo dalla radice costa $O(\lg n)$. Facendo, nel caso peggiore, n estrazioni avremo un costo di $\boldsymbol{O(n \lg n)}$.

Gli ultimi due algoritmi li implementeremo con *min-heap* per semplicità, basandoci sulla definizione di statistiche d'ordine data al paragrafo 2.1.

2.2.2 Lista concatenata

La lista concatenata è una struttura dati in cui ogni elemento è associato a un puntatore contenente l'indirizzo dell'elemento successivo. In questo caso useremo le liste singolarmente concatenate invece di quelle doppiamente concatenate perché per il nostro obiettivo non hanno differenze rilevanti. Una lista singolarmente concatenata ha un nodo testa (*head*) e un puntatore al nodo successivo (*next*), che a sua volta avrà un nodo contente l'elemento e un puntatore al successivo e così via... Il puntatore al successivo dell'ultimo elemento avrà un puntatore a NULL.

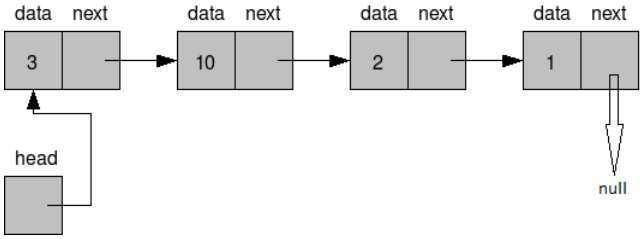


Figura 3: Esempio di lista concatenata singolarmente.

Focalizziamoci sugli esperimenti da compiere, assumiamo con n il numero di elementi/nodi, il **massimo** e il **minimo** vengono calcolati in $\mathbf{O(n)}$ perché i valori inseriti durante la fase di inserimento non vengono ordinati quindi per ricercare il minimo o il massimo dovremmo scorrere tutta la lista fino alla fine. Facendo confronti con il massimo o il minimo attuale, in fase di ispezione.
Invece OS-select e OS-rank verranno eseguiti in $\mathbf{O(n^2)}$. Questo costo è dovuto al peggiore dei casi, ovvero al calcolo del n-esimo valore più piccolo, per forza bruta, basandoci sulla logica dell'algoritmo *Selection-sort*. Questo algoritmo ci permette di dividere in due settori la lista, a sinistra i valori ordinati e a destra i nodi da ispezionare.
Ovviamente tale algoritmo modificherebbe la lista, quindi eseguiamo la ricerca su una lista temporanea in modo tale da non sballare nessun valore e facendo restare invariato il costo previsto
In generale questo algoritmo è molto costoso, ma in questo caso permette una semplice implementazione di OS-select e OS-rank su una lista non ordinata dato che i valori andrebbero comunque ordinati, in qualche modo. Quindi possiamo calcolare il rango o l'i-esimo elemento più piccolo, semplicemente restituendo l'output quando opportuno, troncando la ricerca. Infatti il costo è dovuto al caso peggiore, ma anche atteso.
L'inserimento di nuovi valori avviene in testa.

2.2.3 Lista concatenata ordinata

La lista (singolarmente) concatenata ordinata ha la stessa struttura della lista concatenata non ordinata con l'unica differenza che i valori al suo interno sono **ordinati** in fase di inserimento. Per convenzione li ordiniamo in modo **crescente**, perché in ordine decrescente si comporterebbe in modo speculare.

Quindi adesso possiamo trovare il **minimo** in tempo costante $\mathbf{O(1)}$ perché è il valore contenuto in testa (head). Invece il **massimo** viene calcolato in $\mathbf{O(n)}$ perché il massimo si troverà in fondo alla lista ma non possiamo accederci direttamente tramite un puntatore, quindi siamo costretti a scorrere tutta la lista fino alla fine. Ovviamente se la lista fosse ordinata in modo decrescente i costi saranno simmetrici ai precedenti. I nuovi valori vengono inseriti in ordine durante l'inserimento, ricercando la posizione corretta.

Avendo assunto che i valori sono ordinati in modo crescente, l'**OS-select** ha un costo di $\mathbf{O(n)}$ perché nel peggiore dei casi può essere dato in input n, quindi dovrà restituire il n-esimo elemento. Anche l'**OS-rank** ha un costo di $\mathbf{O(n)}$ perché viene fatta una sorta di "ricerca" contando i valori ispezionati. Quando viene trovato il valore dato in input, restituisco rango (contatore) quindi nel caso peggiore è il n-esimo elemento.

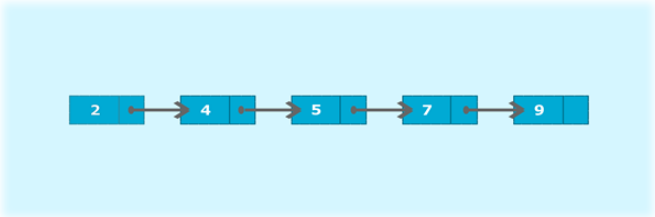


Figura 4: Esempio di lista concatenata ordinata (singolarmente).

3 Descrizione esperimenti

3.1 Introduzione e dati utilizzati

I test, come indicato nel paragrafo 1.2, riguardano diverse implementazioni di algoritmi di statistiche d'ordine dinamiche. Ognuno di essi verrà implementato per ognuna delle tre strutture dati e verranno confrontati i tempi di esecuzione attraverso dei grafici che mostrano gli andamenti dei tempi di esecuzione al crescere del numero di elementi presenti nelle struttura dati.
Come numero di elementi campione (valori di campionamento) prenderemo **valori crescenti linearmente da 100 a 5000 ogni 100**,, ovvero le **dimensioni** delle strutture dati in cui verranno misurati i tempi. **Per ognuno** di questi campioni verranno fatte **150 iterazioni** su cui poi verrà fatta una **media**, in modo tale da avere dei valori più affidabili.

Nello specifico effettueremo prima i test per la ricerca del minimo e del massimo su:

- un *max-heap*
- un *min-heap*
- una lista concatenata
- una lista concatenata ordinata

Poi effettueremo i test degli algoritmi OS-select e OS-rank su:

- un *min-heap*
- una lista concatenata
- una lista concatenata ordinata

Questi ultimi verranno testati solamente per queste strutture dati per i motivi citati nel paragrafo 2.2.1.

Ognuno di questi test verrà eseguito con:

- **valori interi casuali**: nell'intervallo compreso tra 1 e 10000
- **valori interi crescenti**: nell'intervallo compreso tra 1 e il numero di nodi attuale, incrementando di 1
- **valori interi decrescenti**: nell'intervallo compreso tra il numero di nodi attuale e 1, decrementando di 1

I valori casuali cambiano ad ogni iterazione così che il risultato della media dei tempi sia più affidabile. Per ogni algoritmo elencato nel paragrafo 2.1 verrà disegnato un grafico contenente i tempi in funzione del numero di nodi.

Ogni test corrisponde ad un grafico. Quindi in totale avremo, per i primi due algoritmi, 6 grafici contenenti ognuno 4 strutture dati. 3 di questi grafici sono per il massimo, gli altri 3 per il minimo. Ognuno dei 3 grafici è relativo alla tipologia dei valori inseriti, come elencato precedentemente.

Invece per gli altri due algoritmi avremo altri 6 grafici contenenti ognuno 3 strutture dati (*min-heap*, lista ordinata e lista non ordinata). Di cui 3 di questi grafici sono per l'OS-select, gli altri 3 per l'OS-rank. Ognuno dei 3 grafici è relativo alla tipologia dei valori inseriti, come elencato precedentemente.

In totale avremo 12 grafici, perché confrontiamo **4 algoritmi** (sulle strutture dati citate prima) su **3 diversi tipi di input**. Alcuni grafici dovranno essere ingranditi (anche più volte) a causa dell'elevato divario temporale tra le diverse esecuzioni su differenti strutture dati.

Per ogni test saranno salvate le tabelle in file *.tex* con i valori utilizzati per il disegno del grafico.

3.2 Ricerca del minimo

Questi test consistono nel misurare i tempi di esecuzione dell'algoritmo di ricerca del minimo su una lista di valori data in input, per ognuna delle 4 strutture dati elencate nel paragrafo 3.1. Andremo a misurare **solamente** il tempo di esecuzione dell'algoritmo di ricerca del minimo, quindi non misureremo eventuale ordinamenti o ristrutturazioni interne delle strutture dati. Tali modifiche

vengono fatte direttamente all'inserimento del valori, un esempio lo abbiamo nelle liste liste concatenate ordinate che vengono ordinate durante l'inserimento dei valori alla creazione della struttura dati.

Quindi per ogni struttura dati viene registrato il tempo di esecuzione e inserito in una lista per le elaborazioni successive. Questo procedimento si ripete per **3 volte** perché avremo **diversi ordinamenti dei valori** in input (random, crescenti e decrescenti). Alla fine di un singolo test avremo una lista di **4 valori**, che corrisponde al **numero di strutture dati di test**. Invece alla fine di tutti i test (con diverso ordine di input) avremo 3 liste da 4 valori ciascuno, quindi per un totale di 12 misurazioni per numero di elementi campione.
Ad esempio, se eseguo questo test su una lista di valori di 50 elementi, avrò **12 misurazioni totali (per campione**, $4 \cdot 3 = 12$). Questo verrà ripetuto per ogni iterazione e per ogni valore di campionamento citato nel paragrafo 3.1.

Ovviamente le iterazioni servono solamente per aumentare l'affidabilità delle misurazioni, calcolandone la media. Quindi, per esempio, se abbiamo 50 valori di campionamento e alla fine di tutte le misurazioni su un singolo valore campionato abbiamo 12 misurazioni, per il disegnare il grafico di questo esperimento avremo un totale di $50 \cdot 12 = 600$ misurazioni (ovviamente considerando tutte le strutture dati con tutte i tipi di input). Per ogni grafico avremo $50 \cdot 4 = 200$ punti di coordinate cartesiane. Infatti $600/3 = 200$, con 3 le diverse tipologie di input.

3.3 Ricerca del massimo

Questi test consistono nel misurare i tempi di esecuzione dell'algoritmo di ricerca del massimo su una lista di valori data in input, per ognuna delle 4 strutture dati elencate nel paragrafo 3.1. Hanno la **stessa logica e struttura del test del minimo** e otterremo di nuovo **12 misurazioni** per ogni iterazione e per ogni valore di campionamento. Quindi valgono le stesse considerazioni fatte nel paragrafo 3.2

3.4 OS-select

Questi test consistono nel misurare i tempi di esecuzione dell'algoritmo OS-select su una lista di valori data in input, per ognuna delle **3 strutture dati** (no *max-heap*) elencate nel paragrafo 3.1.

Viene generato un valore intero casuale compreso tra 1 e il numero di elementi di test, che sta ad indicare il rango di input per ognuno dei **3 tipi di input**. Vengono misurati i tempi per l'algoritmo sulle 3 diverse strutture dati. Quindi alla fine, per **un** valore di campionamento in una **singola** iterazione, avremo **9 misurazioni** ($3 \cdot 3 = 9$).
Ripeteremo questo per tutte le iterazioni citate al paragrafo 3.1 e per tutti i valori di campionamento. Facendo l'esempio del paragrafo 3.2, nel caso di 6 campioni, alla fine avremo $50 \cdot 9 = 450$ misurazioni (considerando entrambe le strutture dati e i diversi input). Per ogni grafico avremo $50 \cdot 3 = 150$ punti di coordinate cartesiane. Infatti $450/3 = 150$.

3.5 OS-rank

Questi test consistono nel misurare i tempi di esecuzione dell'algoritmo OS-rank su una lista di valori data in input, per ognuna delle **3 strutture dati** elencate nel paragrafo 3.1.

Viene scelto un valore casuale all'interno della lista di valori di input. Questo serve come input all'OS-rank per fornire il suo rango come output.

Ha la **stessa logica e struttura del test dell'OS-select** e otterremo di nuovo **9 misurazioni** per ogni iterazione e per ogni valore di campionamento. Quindi valgono le stesse considerazioni fatte nel paragrafo 3.4.

4 Documentazione del codice

4.1 Librerie utilizzate

Per l'implementazione delle strutture dati sono state usate la librerie:

- per gli heap: *math*, *numpy* e *ABC* (per la classe astratta)
- per le liste (ordinate e non): *random*, per eseguire dei test solamente sulle implementazioni Python

Invece per i test dell'esercizio sono state usate le librerie:

- *matplotlib.pyplot*: per disegnare i grafici
- *numpy*: per il calcolo matriciale per la gestione delle molteplici misurazioni e per gli array
- *pandas*: per la creazione di tabelle
- *jinja2*: di supporto a *pandas* per la creazione di tabelle per Latex
- *random*: per la generazione di valori casuali
- *default_timer* da *timeit*: per la misurazione del tempi

4.2 Analisi delle scelte implementative

In questo paragrafo discuteremo a scopo introduttivo delle principali scelte implementative fatte. Nello specifico saranno trattate nei paragrafi 4.3 e 4.4. Iniziamo col dire che ad ogni tipo di algoritmo viene testato su strutture dati non vuote. Questo perché vengono sempre dati un certo tipo di input, quindi le strutture dati saranno sempre riempite e di conseguenza ci sarà sempre risultato positivo. In questo modo avremo dei risultati più uniformi, finalizzati al confronto delle strutture dati. Inoltre non sarebbe utile al fine dell'esercizio, ovvero quello di confrontare le varie implementazioni di statistiche d'ordine dinamiche sulle diverse strutture dati, in quanto abbiamo bisogno di riempirle per effettuare i confronti.

Ogni algoritmo è implementato nel modo più efficiente e intuitivo per ogni struttura dati, così da confrontare le implementazioni al meglio. L'unica eccezione la facciamo sugli heap, i quali vengono testati anche la ricerca del minimo e del massimo nella struttura **non** dedicata. Ad esempio la ricerca del minimo in un *max-heap*, e viceversa, il massimo per il *min-heap*. Questo è fatto solo a **scopo dimostrativo**. Ovviamente non è molto utile implementare un algoritmo di ricerca del minimo con un *max-heap* e viceversa, come discusso le paragrafo 2.2.1.

Ogni file Python contente l'implementazione di una struttura dati ha dei test interni per verificare il corretto funzionamento delle operazioni principali fornendo degli input e degli output attesi. Questi test **non** vengono eseguiti nei test generali, ma è possibile eseguirli "a parte", eseguendo tali file, per verificare il corretto funzionamento delle strutture dati implementate.

4.2.1 Heap

Gli heap sono implementati tramite liste Python e contengono l'attributo *size* che indica la dimensione (numero di elementi) dell'heap.

Le caratteristiche comuni ad un max-heap e un min-heap vengono implementate in una classe genitore *Heap* che ha due classi figlie, ovvero *Max-heap* e *Min-heap*.

Infatti con l'ereditarietà e il polimorfismo di Python possiamo avere del codice non ridondante, soprattutto per quanto riguarda la ricerca del massimo e del minimo, che a seconda del tipo di Heap viene fatta in modo differente. Così come i metodi *heapify* ed *editKey*, che variano le implementazioni a seconda che sia un max-heap o min-heap. Anche per le funzioni *maximum* e *minimum* avremo diverse implementazioni a seconda del tipo di heap per motivi strutturali (paragrafo 2.2.1).Nel paragrafo 4.3 verrà spiegato più in dettaglio.

4.2.2 Liste concatenate (ordinate e non)

Le liste concatenate sono collegate singolarmente e non doppiamente perché in questo caso non ci sono differenze rilevanti, a vantaggio della semplicità strutturale.

Le liste (ordinate e non) vengono implementate utilizzando puntatori ai nodi. Un singolo nodo è rappresentato con una classe *Node* che avrà i campi *value* e *next*.

L'ereditarietà e il polimorfismo di Python vengono usati anche qui, in modo analogo a quanto spiegato nel paragrafo 4.2.1. Quindi avremo una classe genitore *LinkedList* con una classe figlia *SortedLinkedList* che eredita le proprietà comuni alla lista non ordinata. Ovviamente varierà l'implmentazione dell'inserimento, dato che è la differenza principale tra i due tipi di lista concatenata. Infatti la classe figlia ha il metodo *addElement* sovrascritto. Questo perché gli elementi della lista vengono inseriti in ordine crescente all'inserimento.

Inoltre una lista non ordinata avrà i metodi OS-rank e OS-select implementati in modo differente dato che la lista deve essere ordinata, tenendo conto delle considerazioni fatte nel paragrafo 2.2.2. Ovviamente il metodo *minimum* per la ricerca del minimo avrà una differente implementazione nelle due liste. Maggiori dettagli verranno descritti nel paragrafo 4.3.

4.3 Diagrammi UML delle implementazioni

4.3.1 Implementazione heap

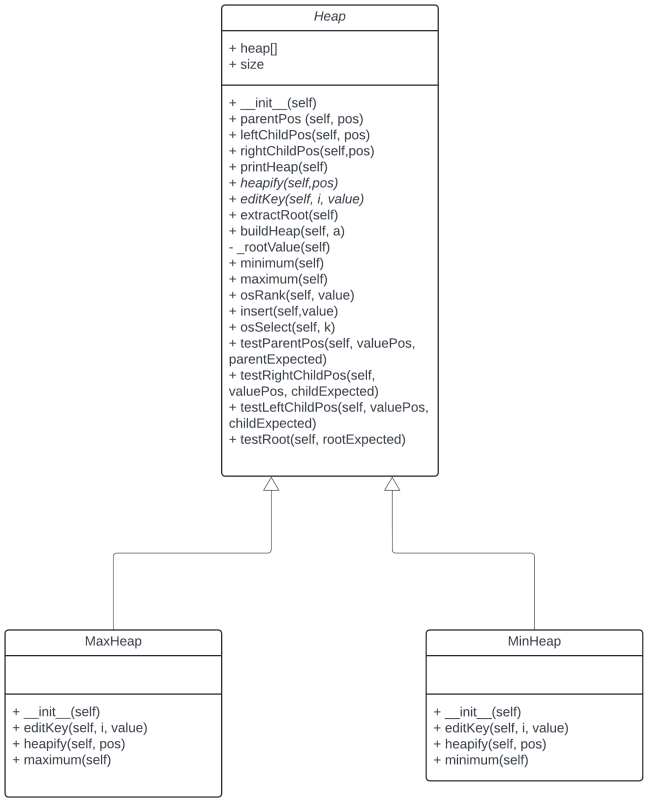


Figura 5: Diagramma UML dell'implementazione degli heap.

Come possiamo vedere dalla figura 5, la classe genitore *Heap* funge da classe astratta perché i metodi *heapify* e *editKey* richiedono delle implementazioni specifiche nelle classi figlie. Infatti tali metodi verranno sovrascritti.

Tutte le implementazioni degli altri metodi vengono ereditate dalle due classi figlie. I metodi *maximum* e *minimum*, nonostante abbiano già l'implementazione (per forza bruta) nella classe genitore, vengono sovrascritti nelle classi figlie a seconda del tipo di heap. In questo modo possiamo ottimizzare un determinato algoritmo grazie alla proprietà della struttura dati sulla quale è eseguito.

Nella classe genitore sono presenti i campi *heap[]* e *size* che rappresentano rispettivamente la lista Python (per semplicità di utilizzo) contenente l'heap. Essi verranno ereditati dalla classi figlie.

I metodi *osSelect* e *osRank* sono implementati nella classe genitore perché da un punto di vista teorico potrei voler ricercare i valori più grandi, anziché i più piccoli. Noi tratteremo solo i più piccoli basandoci sulla definizione di statistica d'ordine descritta nel paragrafo 2.1.

4.3.2 Implementazione liste

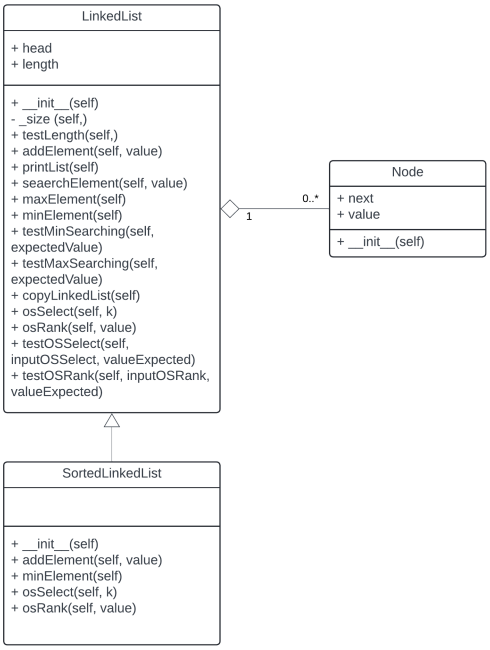


Figura 6: Diagramma UML dell'implementazione delle liste (ordinate e non).

Riferendoci alla figura 6, possiamo vedere che la classe *LinkedList* (per le liste **non** ordinate) è la classe genitore della sua classe figlia *SortedLinkedList* (per le liste ordinate). La classe genitore, e di conseguenza anche la figlia, hanno una classe aggregata *Node* che rappresenta un **singolo nodo** della lista. Quindi, appunto, la classe figlia e la classe genitore aggregano più nodi formando la lista.

Nella classe genitore sono presenti i campi *head* e *length* che sono rispettivamente il puntatore alla testa della lista (inizialmente nullo) e la lunghezza della lista (inizialmente 0). Essi vengono ereditati nella classe figlia.

La classe genitore **non** è una classe astratta infatti è istanziabile, così come ogni suo metodo è dotato di implementazione.

Facendo riferimento alla logica strutturale descritta nel paragrafo 4.3.1, anche adesso, ogni metodo della classe figlia viene (avente lo stesso nome) sovrascritto con l'implementazione che segue la logica delle liste ordinate. Le implementazioni di ogni metodo sono descritte nel paragrafo 4.4. Ovviamente quando viene aggiunto un elemento, il nuovo nodo viene creato e riempito con il valore ed il puntatore al successivo corretto.

4.4 Descrizione dei metodi implementati

I successivi sub-paragrafi sono relativi ai 3 file Python:

- implementazioni heap (*heap_implementaiton.py*)
- implementazioni liste (*lists_implementation.py*)
- test richiesti dall'esercizio (*tests.py*)

Per ognuno di essi saranno elencate le classi con i relativi metodi e campi.

4.4.1 Implementazioni heap

Le funzioni *testHeap()* , *testMaxAndMinHeaps(LinkedList)* e *testOSselectOSrankMinHeap(LinkedList)* servono per verificare il corretto funzionamento delle strutture dati confrontando gli output con valori attesi. Non fanno parte di nessuna classe e non vengono usate/chiamate dai test dell'esercizio.

La funzione *swap(a,b)* serve a scambiare due valori.
Le classi sono indicate con un punto, i metodi con un trattino.

- **Heap:** *heap* (lista con i valori dell'heap) e *size* (dimensione heap) sono i campi
  - **parentPos(pos)**: restituisce la posizione del genitore, data una posizione di un valore
  - **leftChildPos(pos)**: restituisce la posizione del figlio sinistro, data una posizione di un valore
  - **rightChildPos(pos)**: restituisce la posizione del figlio destro, data una posizione di un valore
  - **printHeap()**: stampa l'heap
  - **heapify(pos)**: mantiene le proprietà dell'heap. L'implementazione viene lasciata (passata) ai alle classi figlie (*MaxHeap* e *MinHeap*.
  - **editKey(i,value)**: modifica un valore dell'heap. Con i l'indice della chiave da modificare. L'implementazione viene lasciata (passata) ai alle classi figlie (*MaxHeap* e *MinHeap*
  - **extractRoot()**: estrae la radice dell'heap restituendola
  - **buildHeap()**: costruisce un heap a partire da una lista. Utilizza l'*heapify* polimorfico. Usato nei test per creare gli heap.
  - **_rootValue()**: restituisce la radice dell'heap
  - **maximum()**: cerca il massimo nelle foglie dell'heap e lo restituisce.
  - **minimum()**: cerca il minimo nelle foglie dell'heap e lo restituisce.
  - **osRank(value)**: Dato un valore, restituisce il suo rango. Conta quante estrazioni di radice vengono fatte e quando trova il valore di input restituisce il contatore (rango)
  - **osSelect(k)**: Dato il rango, restituisce il k-esimo valore più piccolo nella lista. Compiendo k estrazioni di radice.
  - **insert(value)**: inserisce un valore nell'heap. Viene messo alla fine e aggiustato da *editKey(i,value)*. Non usato per i test, perché meno pratico di *buildHeap*.
  - **testParentPos(valuePos,parentExpected)**: controlla se la posizione del padre è quella attesa. Stampa un messaggio di successo o insuccesso. Usato per verifica, test interno.
  - **testRightChildPos(valuePos,childExpected)**: controlla se la posizione del figlio destro è quella attesa. Stampa un messaggio di successo o insuccesso. Usato per verifica, test interno.
  - **testLeftChilPdos(valuePos,childExpected)**: controlla se la posizione del figlio sinistro è quella attesa. Stampa un messaggio di successo o insuccesso. Usato per verifica, test interno.
- **MaxHeap:** eredita i campi della classe genitore
  - **maximum()**: override. Restituisce il massimo, ovvero la radice dell'heap.
  - **heapify(pos)**: override, mantiene le proprietà del max-heap. Porta gli elementi più grande verso la radice.
  - **editKey(i,value)**: override, modifica un valore del max-heap solo se il valore è maggiore della chiave esistente. Con i l'indice della chiave da modificare.
- **MinHeap:** eredita i campi della classe genitore
  - **maximum()**: override. Restituisce il massimo, ovvero la radice dell'heap.
  - **heapify(pos)**: override, mantiene le proprietà del min-heap. Porta gli elementi più piccoli verso la radice.
  - **editKey(i,value)**: override, modifica un valore del min-heap solo se il valore è minore della chiave esistente. Con i l'indice della chiave da modificare.

4.4.2 Implementazioni liste

Le funzioni *testLinkedList()* , *testOSSortedLinkedList(LinkedList)* e *testMaxAndMinSearch(LinkedList)* servono per verificare il corretto funzionamento delle strutture dati confrontando gli output con valori attesi. Non fanno parte di nessuna classe e non vengono usate/chiamate dai test dell'esercizio. Le classi sono indicate con un punto, i metodi con un trattino.

- **Node:** *value* e *next* (puntatore al nodo successivo) sono i campi
- **LinkedList:** *head* (puntatore alla testa) e *length* (lunghezza lista) sono i campi
  - **_size()**: restituisce la lunghezza della lista. Usata solo per verifica, test interno.
  - **testLength()**: stampa messaggio di successo o insuccesso. Utilizzato per verifica, test interno.
  - **addElement(value)**: inserisce in testa un nodo (creandolo) nella lista.
  - **printList()**: stampa la lista dalla testa alla fine.
  - **searchElement(value)**: ricerca un valore nella lista. Ritorna un valore booleano. Usato per verifica, test interno.
  - **maxElement()**: restituisce il massimo valore presente nella lista. Viene cercato per forza bruta.
  - **minElement()**: restituisce il minimo valore presente nella lista. Viene cercato per forza bruta.
  - **testMinSearching(expectedValue)**: stampa messaggio di successo o insuccesso. Controlla se il minimo è quello atteso. Usato per verifica, test interno.
  - **testMaxSearching(expectedValue)**: stampa messaggio di successo o insuccesso. Controlla se il massimo è quello atteso. Usato per verifica, test interno.
  - **copyLinkedList()**: copia il contenuto della lista di riferimento in una nuova lista di tipo LinkedList (non ordinata), restituendola
  - **osSelect(k)**: dato il rango, restituisce il k-esimo valore più piccolo nella lista. Basato su *Selection-sort*.
  - **osRank(value)**: dato il valore, restituisce il suo rango. Basato su *Selection-sort*.
  - **testOSSelect(inputOSSelect,valueExpected)**: controlla se l'output dell'OS-select è quello atteso. Stampa un messaggio di successo o insuccesso. Usato per verifica, test interno.
  - **testOSRank(inputOSRank,valueExpected)**: controlla se l'output dell'OS-rank è quello atteso. Stampa un messaggio di successo o insuccesso. Usato per verifica, test interno.
- **SortedLinkedList:** *head* (puntatore alla testa) e *length* (lunghezza lista) sono i campi ereditati
  - **addElement(value)**: override. Inserisce un nodo (creandolo) nella lista, mantenendo l'ordine crescente.
  - **minElement()**: override. Restituisce il valore della testa, se presente.
  - **osSelect(k)**: override. Dato il rango, restituisce il k-esimo valore più piccolo nella lista, ispezionando k nodi a partire dalla testa.
  - **osRank(value)**: override. Dato il valore, restituisce il suo rango. Ispezionando ogni nodo affinché non trova il valore, restituendo il contatore (rango).

4.4.3 Test

In questo file Python non ci sono classi perchè vengono eseguiti solamente i test chiesti dall'esercizio. Quindi importiamoi file (le implementazioni) e le librerie (paragrafo 4.1) necessarie per svolgere i test.

I test vengono effettuati solamente sul metodo da testare. Adesso elenchiamo tutte le funzioni necessarie ai test. Sono presenti una funzione per ogni tipologia di test per ogni struttura dati. Questo è utile per avere un codice non ridondante e logicamente lineare ed intuitivo. Adesso indichiamo con un punto tutte le funzioni.

Generazione valori (diverse tipologie di input: random, crescenti e decrescenti):

- **generateRandomValues(totalValue, minRandomValue, maxRandomValue)**: genera e restituisce una lista di valori interi casuali basati sui parametri in ingresso
- **generateIncreasingValues(minValue, maxValue)**: genera e restituisce una lista di valori interi in ordine crescente basata sui parametri in ingresso.
- **generateDecreasingValues(maxValue, minValue)**: genera e restituisce una lista di valori interi in ordine decrescenti basata sui parametri in ingresso.
- **searchMinMinHeapTest(values)**: misura e restituisce il tempo di esecuzione della ricerca del minimo in un min-heap. Riceve in input la lista di valori per creare la struttura dati e riempirla, ovviamente ciò non fa parte del test.

Misurazioni per algoritmo, per ogni struttura dati:

- **searchMinMaxHeapTest(values)**: misura e restituisce il tempo di esecuzione della ricerca del minimo in un max-heap. Riceve in input la lista di valori per creare la struttura dati e riempirla, ovviamente ciò non fa parte del test.
- **searchMaxMaxHeapTest(values)**: misura e restituisce il tempo di esecuzione della ricerca del massimo in un max-heap. Riceve in input la lista di valori per creare la struttura dati e riempirla, ovviamente ciò non fa parte del test.
- **searchMaxMinHeapTest(values)**: misura e restituisce il tempo di esecuzione della ricerca del massimo in un min-heap. Riceve in input la lista di valori per creare la struttura dati e riempirla, ovviamente ciò non fa parte del test.
- **osRankMinHeapTest(values, targetValue)**: misura e restituisce il tempo di esecuzione del OS-rank in un min-heap. *targetValue* è l'input dell'OS-rank. Riceve in input la lista di valori (*values*) per creare la struttura dati e riempirla, ovviamente ciò non fa parte del test.
- **osSelectMinHeapTest(values, iPos)**: misura e restituisce il tempo di esecuzione del OS-select in un min-heap. *iPos* è l'input dell'OS-select generato. Riceve in input la lista di valori (*values*) per creare la struttura dati e riempirla, ovviamente ciò non fa parte del test.
- **searchMaxSLLTest(values)**: misura e restituisce il tempo di esecuzione della ricerca del massimo in una lista concatenata ordinata. Riceve in input la lista di valori per creare la struttura dati e riempirla, ovviamente ciò non fa parte del test.
- **searchMaxLLTest(values)**: misura e restituisce il tempo di esecuzione della ricerca del massimo in una lista concatenata non ordinata. Riceve in input la lista di valori per creare la struttura dati e riempirla, ovviamente ciò non fa parte del test.
- **searchMinSLLTest(values)**: misura e restituisce il tempo di esecuzione della ricerca del minimo in una lista concatenata ordinata. Riceve in input la lista di valori per creare la struttura dati e riempirla, ovviamente ciò non fa parte del test.
- **searchMinLLTest(values)**: misura e restituisce il tempo di esecuzione della ricerca del minimo in una lista concatenata non ordinata. Riceve in input la lista di valori per creare la struttura dati e riempirla, ovviamente ciò non fa parte del test.
- **osRankSLLTest(values, targetValue)**: misura e restituisce il tempo di esecuzione di OS-rank in una lista concatenata ordinata. Riceve in input la lista di valori per creare la struttura dati e riempirla, ovviamente ciò non fa parte del test. *targetValue* è l'input dell'OS-rank.
- **osRankLLTest(values, targetValue)**: misura e restituisce il tempo di esecuzione di OS-rank in una lista concatenata non ordinata. Riceve in input la lista di valori per creare la struttura dati e riempirla, ovviamente ciò non fa parte del test. *targetValue* è l'input dell'OS-rank.
- **osSelectSLLTest(values, iPos)**: misura e restituisce il tempo di esecuzione di OS-select in una lista concatenata ordinata. Riceve in input la lista di valori per creare la struttura dati e riempirla, ovviamente ciò non fa parte del test. *iPos* è l'input dell'OS-select.
- **osSelectLLTest(values, iPos)**: misura e restituisce il tempo di esecuzione di OS-select in una lista concatenata non ordinata. Riceve in input la lista di valori per creare la struttura dati e riempirla, ovviamente ciò non fa parte del test. *iPos* è l'input dell'OS-select.

Disegna grafici e salva tabelle su file:

- **createAndSave Table(xValues, timesPerDataStructure, fileName)**: crea e salva su un file la tabella si un test effettuato su una tipologia di input (con tutte le strutture dati). Utilizzate per le tabelle dei paragrafi successivi. Inoltre viene anche impostata la precisione dei risultati, in questo caso 5. Riceve in input i valori campione (le ascisse),la matrice contenente i valori per una singola tipologia di input (ad esempio valori random...), e il nome del file su cui salvare i dati. Le tabelle vengono salvate in formato *.tex*
- **drawGraph(graphTitle, xValues, testFinalTimes, zoom=False,zoomGrade=20)**: disegna e restituisce un grafico (senza salvarlo). Riceve in input il titolo, le ascisse (valori campione), la matrice contenente i risultati per tipologia di input. Poi i parametri *zoom* e *zoomGrade* sono impostati di default, se abilitati verrà generato un grafico ingrandito di un certo fatto dato in input (*zoomGrade*)
- **saveGraph(nameFile, graph)**: salva l'immagine del grafico dato in input con il nome di *nameFile*

Funzioni per la gestione dei risultati per ogni tipologia di input e per ogni algoritmo:

- **searchMaxTests(values)**: raggruppa tutti i tempi di esecuzioni misurati per ogni struttura dati di test per ogni tipologia di input in una lista di lista per un singolo algoritmo, in questo caso la ricerca del massimo. Restituisce questa lista al chiamante. Riceve in input le 3 diverse tipologie di input su cui eseguire i test. Praticamente questa funzione funge da "amministratore" dei risultati per ogni test (su differenti algoritmi).
- **searchMinTests(values)**: raggruppa tutti i tempi di esecuzioni misurati per ogni struttura dati di test per ogni tipologia di input in una lista di lista per un singolo algoritmo, in questo caso la ricerca del minimo. Restituisce questa lista al chiamante. Riceve in input le 3 diverse tipologie di input su cui eseguire i test. Praticamente questa funzione funge da "amministratore" dei risultati per ogni test (su differenti algoritmi).
- **oSRankTests(values)**: stessa logica dei precedenti. Stesso input e output dei precedenti.
- **oSSelectTests(values)**: stessa logica dei precedenti. Stesso input e output dei precedenti.

Funzione principale da cui partono tutte le chiamate alle altre funzioni:

- **runAllTests()**: si comporta da "*main*". Effettua i calcoli matriciali delle iterazioni calcolandone la media, in modo da fornire il quantitativo corretto e affidabile di risultati alla funzione *drawGraph*. In sintesi si occupa di chiamare tutte le funzioni necessarie alla creazione dei grafici.

4.5 Alcuni snippets rilevanti

I commenti #... stanno ad indicare che in quella zona è presente altro codice ma che non è rilevante nello snippet.

4.5.1 Heap

```
class Heap:
    def __init__(self):
        self.size = 0
        self.heap = []

    #...
    #...

    def heapify(self,pos):
        pass

    #...
    #...

    #return max or min value of the heap
    def _rootValue(self):
        if self.size > 0:
            return self.heap[0]
```

```
        else:
            print("Heap is empty")
            return

    #...
    #...

    def maximum(self):
        if self.size > 0:
            middle = math.ceil(self.size/2)
            maximum = self.heap[middle]
            for i in range(middle,self.size):
                if self.heap[i] > maximum:
                    maximum = self.heap[i]
            return maximum
        else:
            print("Heap is empty")
            return

    #...
    #...
```

Frammento 1: implementazione della classe genitore Heap. Sono presenti solamente alcuni metodi rilevanti per brevità

Come possiamo vedere dal frammento 1, come già accennato nei paragrafi precedenti, i metodi *heapify* (riga 9), *editKey* (non presente perché non usato),*maximum* (riga 15) e *minimum* (presente nel sorgente, ma omesso nel frammento per brevità) verranno sovrascritti nelle classi figlie. Soffermiamoci su *heapify* e *maximum* per un max-heap.

```
class Maxheap(Heap):
    #...
    #...

    def heapify(self, pos):
        left = self.leftChildPos(pos)
        right = self.rightChildPos(pos)
        if (left < self.size) and (self.heap[left] > self.heap[pos]):
            maxValuePos = left
        else:
            maxValuePos = pos
        if (right < self.size) and (self.heap[right] > self.heap[maxValuePos]):
            maxValuePos = right
        if maxValuePos != pos:
            self.heap[pos],self.heap[maxValuePos] = swap(self.heap[pos],self.
    heap[maxValuePos])
            self.heapify(maxValuePos)

    def maximum(self):
        return super()._rootValue()
#...
#...
```

Frammento 2: implementazione della classe figlia MaxHeap. Sono presenti solamente alcuni metodi sovrascritti rilevanti per brevità

Come vediamo dal frammento 2, la classe *MaxHeap* è figlia di *Heap* quindi eredita tutti i suoi metodi sovrascrivendone alcuni, come quelli qui presenti. Uno tra questi è *heapify* che ha una implementazione nel caso di un max-heap e un'altra nel caso di un min-heap. Infatti nella classe *MinHeap* avremo un'altra implementazione sovrascritta a quella genitore (rispetto alla classe Heap). Lo stesso vale per la classe *maximum*, la quale restituirà il valore della radice (metodo presente nella classe genitore) nel caso di un max-heap. Se invece volesi cercare il minimo in un max-heap, dovremmo usare il metodo **non** sovrascritto per motivi strutturali della classe genitore *Heap*, cercandolo nelle foglie.
Nella classe Minheap, figlia di Heap, con la solita logica di quella nel frammento 2 avremo i soliti metodi elencati sovrascritti (implementati in modo che soddisfino le proprietà del min-heap) con l'unica differenza che faremo un override di *minimum* e **non** di *maximum*.

Adesso soffermiamoci sulle implementazioni di OS-select e OS-rank di un heap, come esposto nel paragrafo 2.2.1.

```
class Heap:
    #...
```

```
    #...

    def extractRoot(self):
        if self.size < 1:
            return None
        rootValue = self.heap[0]
        self.heap[0] = self.heap[self.size-1]
        self.size -= 1
        self.heap = self.heap[:self.size]
        self.heapify(0)
        return rootValue

    #...
    #...

    #extracts the root, so is not re-usable
    def osRank(self, value):
        tmpRank = 0
        root = self.extractRoot()
        while tmpRank < self.size:
            if root == value:
                return tmpRank+1
            else:
                tmpRank += 1
                root = self.extractRoot()
        return 0

    #...
    #...
    def osSelect(self,k):
        if self.size <= 0 or k < 1 or k > self.size:
            return None
        for i in range(k):
            root = self.extractRoot()
        return root
#...
#...
```

Frammento 3: implementazioni di OS-select e OS-rank per Heap

Nel frammento 3, vediamo le implementazioni di OS-select e OS-rank. I test verranno fatti su dei **min-heap**, tenendo conto delle considerazioni fatte nel paragrafo 2.2.1. Nel paragrafo 4.4.1 vengono descritti brevemente i metodi.

4.5.2 Liste

Adesso guardiamo alcuni snippet rilevanti riguardanti l'implementazione delle liste (ordinate e non).

```
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None
```

Frammento 4: Implementazione classe Node

Nel frammento 4 vediamo l'implementazione della classe *Node* che rappresenta un **singolo** nodo della lista (ordinata o non).

```
class LinkedList:
    def __init__(self):
        self.head = None
        self.length = 0
    #...
    #...
    def addElement(self, value):
        newNode = Node(value)
        newNode.next = self.head
        self.head = newNode
        self.length += 1
    #...
    #...
    def minElement(self):
        if not self.head:
            return None
        min = self.head.value
        currentNode = self.head.next
```

```
        while currentNode:
            if currentNode.value < min:
                min = currentNode.value
            currentNode = currentNode.next
        return min
    #...
    #...
#...
#...
```

Frammento 5: Implementazione della classe LinkedList con alcuni suoi metodi rilevanti

Infatti come vediamo dal frammento 5 questa classe aggregherà più nodi formando la lista. Partendo dal campo *head*, presente nella classe LinkedList.
Inoltre possiamo vedere l'implementazione del metodo *addElement* che inserisce in testa alla lista un nuovo nodo (creandolo).
Un'altra implementazione importante da sottolineare è quella di *minElement*. Infatti nel frammento 5 vediamo quella relativa ad una lista **non ordinata**. Quindi trova il minimo per forza bruta, confrontando il minimo attuale trovato via via durante l'ispezione.
Nel frammento 7 vedremo l'implementazione sovrascritta per la lista ordinata.
Andando avanti con metodi importanti presenti nella classe *LinkedList*, guardiamo il metodo *osSelect* nel frammento 6. Il metodo *osRank* non lo vediamo perché è basato sulla solita logica, ma ovviamente rispettando il fine di tale metodo.

```
    #...
    #...
    def osSelect(self,k):
        if k > self.length or k <= 0:
            return None

        # Copy list nodes into another list
        copyList = self.copyLinkedList()

        rank = 0
        # selection-sort based
        currentNode = copyList.head
        while currentNode:
            minNode = currentNode
            nextNode = currentNode.next
            while nextNode:
                if nextNode.value < minNode.value:
                    minNode = nextNode
                nextNode = nextNode.next
            if minNode != currentNode:
                currentNode.value, minNode.value = minNode.value, currentNode.
    value
            rank += 1
            if rank == k:
                return currentNode.value
            currentNode = currentNode.next
        return None
    #...
    #...
#...
#...
```

Frammento 6: Implementazione metodo osSelect della classe LinkedList

Nel frammento 6 vediamo l'implementazione dell'algoritmo OS-select per una lista non ordinata. Come già accennato nel paragrafo 2.2.2, questa implementazione è basata sull'algoritmo *Selection-sort* infatti notiamo la struttura molto simile. Ovviamente applicando le modifiche per renderlo utile al nostro scopo, ovvero quello di far restituire il k-esimo valore più piccolo.
Nelle righe 19-20 viene restituito il valore all'interno del nodo quando è stato trovato, interrompendo l'ordinamento della lista di copia. Tale algoritmo non modificherà la lista agendo su una lista temporanea, come spiegato al paragrafo 2.2.2.

Adesso passiamo alle liste ordinate:

```
class SortedLinkedList(LinkedList):
    #...
    #...

```

```
    # Adds elements in grow way
    def addElement(self, value):
        newNode = Node(value)
        self.length += 1
        if not self.head or value < self.head.value:
            newNode.next = self.head
            self.head = newNode
        else:
            currentNode = self.head
            while currentNode.next and currentNode.next.value < value:
                currentNode = currentNode.next
            newNode.next = currentNode.next
            currentNode.next = newNode

    # Returns min element from sorted list (growing way)
    def minElement(self):
        if self.head is None:
            return None
        else:
            return self.head.value

    # Returns k-th smallest element from sorted list
    def osSelect(self,k):
        if k > self.length:
            return None
        currentNode = self.head
        i = 1
        while i < k and currentNode:
            i += 1
            currentNode = currentNode.next
        return currentNode.value
    #...
    #...
#...
#...
```

Frammento 7: Implementazione classe figila SortedLinkedList

Nel frammento 7 vediamo la classe figlia di LinkedList che eredita tutti i suoi metodi tranne quelli sovrascritti, che vediamo. Notiamo subito una notevole differenza nell'implementazione sovrascritta del metodo *addElement*. Infatti esso si occupa di inserire **nella giusta posizione** il valore dato in input.
Anche il metodo *minElement* viene sovrascritto ed ha un'implementazione che consente un costo costante nella ricerca del minimo come accennato nel paragrafo 2.2.3.
Infine anche *osSelect* viene sovrascritto, così come osRank (non presente nello snippet per brevità). Infatti l'implementazione di osSelect sovrascritta permette la ricerca dell'k-esimo valore più piccolo in tempo lineare perché la lista è già ordinata in modo crescente all'inserimento.

4.5.3 Test

Adesso soffermiamoci su alcuni snippets rilevanti per la comprensione del funzionamento e della struttura dei test effettuati.
Partiamo dalla funzione principale *runAllTests* situanta nel file *tests.py*. Vediamo il primo snippet, omettendo alcune parti non rilevanti:

```
    x = [g for g in range(100,5000+1,100)]
    finalTimesMaxTest = np.array([])
    finalTimesMinTest = np.array([])
    finalTimesOSSelectTest = np.array([])
    finalTimesOSRankTest = np.array([])
    for j in x:
        timesMaxTests = []
        timesMinTests = []
        timesOSSelectTests = []
        timesOSRankTests = []
        for i in range(nIterations):
            values = []
            values.append(generateRandomValues(j, 1, 10000))
            values.append(generateIncreasingValues(1, j))
            values.append(generateDecreasingValues(j,1))
            print("\nTest: ",i+1,"with ",j," elements")
            # Maximum
            timesMaxTest = searchMaxTests(values)
            timesMaxTests.append(timesMaxTest)
```

```
20              # Minimum
21              timesMinTest = searchMinTests(values)
22              timesMinTests.append(timesMinTest)
23              # OS-Select
24              timesOSSelectTest = oSSelectTests(values)
25              timesOSSelectTests.append(timesOSSelectTest)
26              # OS-Rank
27              timesOSRankTest = oSRankTests(values)
28              timesOSRankTests.append(timesOSRankTest)
29
30          # Calculate averages:
31
32          # Maximum
33          timesMaxTests = np.sum(timesMaxTests, axis=0)
34          timesMaxTests = np.divide(timesMaxTests, nIterations)
35          finalTimesMaxTest = np.append(finalTimesMaxTest, timesMaxTests)
36          #...
37          #...
38          #Altri calcoli basati sulla solita logica...
```

Frammento 8: frammento della funzione runAllTests di tests.py

Come possiamo vedere dal frammento 8, per ogni valore (linea 6) di campionamento presente nella lista x (linea 1) eseguiamo le 150 iterazioni (linea 11) in modo tale da stabilizzare i dati. Tra la riga 12 e la riga 15 vengono generate 3 liste contenenti le tre diverse tipologie di valori da dare in input alle strutture dati per testare gli algoritmi, come descritto nel paragrafo 3.1.

Dalla riga 17 alla riga 28 notiamo un meccanismo di calcolo simile, ma per diverse strutture dati. Quindi analizziamo solamente la prima perché le altre sono simili. Riferendoci alle righe 18 e 19 del frammento 8 , possiamo vedere che la funzione *searchMaxTests* che restituisce una lista di liste salva questi valori in *timesMaxTest*. Ovvero i risultati si una **singola iterazione** del test di massimo (in questo caso), eseguito su tutte le tipologie di input elencate nel paragrafo 3.1. Poi alla riga 19 salva i risultati della singola iterazione in una lista, la quale ci servirà successivamente per il calcolare la media per tutte le *nIterations*. Ripetendo questo meccanismo per 150 volte.

Adesso entriamo nella funzione *searchMaxTests* per capire che cosa succede al suo interno:

```
1 def searchMaxTests(values):
2     times = []
3     for i in range(len(values)):
4         timeMaxHeap = searchMaxMaxHeapTest(values[i])
5         timeMinHeap = searchMaxMinHeapTest(values[i])
6         timeSLL = searchMaxSLLTest(values[i])
7         timeLL = searchMaxLLTest(values[i])
8         times.append([timeMinHeap, timeLL, timeSLL, timeMaxHeap])
9     return times
```

Frammento 9: implementazione della funzione searchMaxTests

Come vediamo dal frammento 9, questa funzione esegue per la lunghezza della lista *values* (ovvero 3) tutti i test di ricerca del massimo per ogni struttura dati interessata. Le iterazioni di *values* seguono l'ordine di generazione di essa, ovvero quello delle righe 13,14,15 del frammento 8: random, crescenti, decrescenti. Nel frammento 9 salviamo i risultati di ogni test (effettuati su ognuna delle strutture dati) in una lista di 4 elementi che a sua volta farà parte di una lista di 3 elementi (ovvero la lunghezza di *values*). Quest'ultima sarà restituita al chiamate e verrà salvata in *timesMaxTest* del frammento 8. Questa logica è usata in tutti i test riguardanti i 4 diversi algoritmi (massimo, minimo, OS-select e OS-rank). Ovviamente varieranno le dimensioni delle liste sulla base di quanto descritto nel paragrafo 3.1.

Ora entriamo in un metodo qualsiasi tra questi quattro, perché gli altri hanno la stessa logica implementativa. Ad esempio andiamo nel metodo *searchMaxSLLTest*, ovvero quello che si occupa della misurazione dei tempi per la ricerca del massimo in una lista ordinata.

```
1 def searchMaxSLLTest(values):
2     newSortedList = LL.SortedLinkedList()
3     for i in values:
4         newSortedList.addElement(i)
5
6     start = timer()
7     maxSLL = newSortedList.maxElement()
8     end = timer()
```

```
9       time = end - start
10      return time
```

Frammento 10: misurazione della ricerca del massimo in una lista concatenata ordinata

Come possiamo vedere dallo snippet 10, abbiamo in input i valori si una **singola** tipologia, ad esempio **solamente** random. Alla riga 2 creiamo un nuova lista ordinata che riempiremo alle righe successive. Effettuiamo la misurazione tramite una libreria esterna, citata nel paragrafo 4.1. Restituendo al chiamante il tempo impiegato nell'esecuzione della **sola** ricerca del massimo. Questa logica è usata in per ogni struttura dati e per ogni tipo di test.

Tornando al frammento 8, adesso dobbiamo fare la media di tutti i risultati ottenuti durante le 150 iterazioni. Quindi alla riga 33 viene eseguita la somma matriciale con "**axis=0**" ovvero la somma eseguite su un asse specifico, le righe in questo caso.

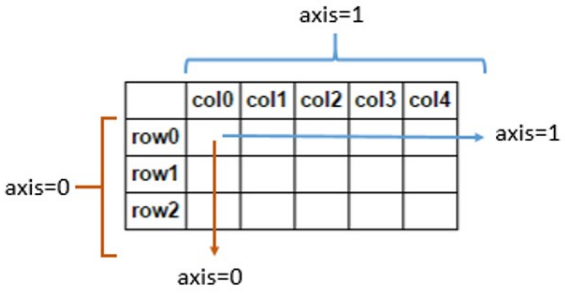


Figura 7: Esempio grafico per la somma lungo un asse specifico

Come vediamo dalla figura 7, eseguiremo la somma lungo le righe. Per comprendere meglio il funzionamento di questa somma che apparentemente risulta complicata ma ci permette di risparmiare molte variabili nello sviluppo del codice, guardiamo la seguente immagine.

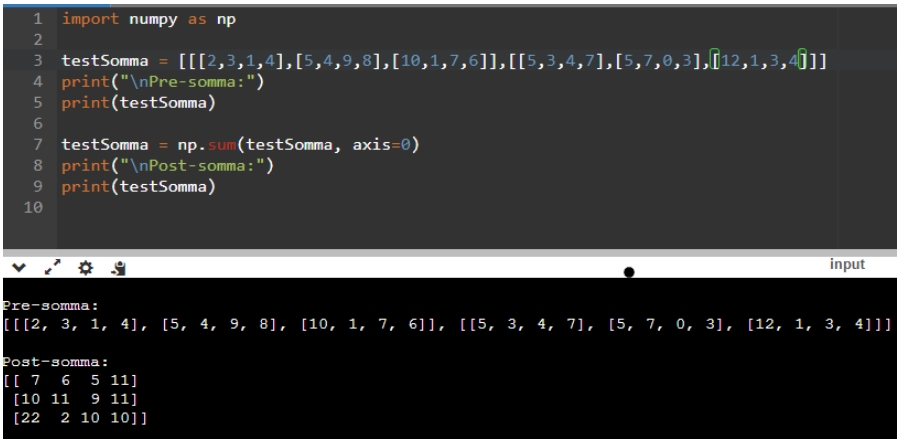


Figura 8: Esempio di somma matriciale riferita all'asse 0, con relativo output di esecuzione. Attenzione: questo codice **non** è presente nei test ma serve solo per far comprendere come opera questa somma, con un input molto simile a quello dei test, ad esempio quello del massimo.

Nella figura 8, i valori inseriti a scopo dimostrativo non sono importanti ma la lunghezza delle liste lo è. Infatti la lunghezza delle liste contente i numeri è 4, ovvero la **stessa** della funzione del frammento 9, che a sua volta sono contenute in una lista formata da 3 liste. Ciò ci aiuta a comprendere come opera la somma avendo una lista molto simile a quella contenente i veri risultati. Alla fine delle iterazioni (per un singolo valore campione), nel frammento 8, avremo che la lista avrà lunghezza 150, ovvero 150 liste da 3 liste contenenti ognuno **4** valori. Quindi dall'output dell'immagine 8 vediamo che ogni valore viene sommato in base alla posizione in cui si trova, riferito alle righe. Ad esempio $2 + 5 = 7$, come primo elemento dell'esempio in figura, e così via... Quindi alla fine ci troveremo tutti i valori sommati con i giusti valori relativi alla propria struttura dati e alla corretta tipologia di dati sulla quale sono stati effettuati i soliti test.

Adesso torniamo al codice del frammento 8. Siccome stavamo calcolando la media, alla riga 34 divideremo tutti i risultati ottenuto per il test di massimo per il numero di iterazioni *nIterations*, in questo caso 150.

Successivamente, alla riga 35 aggiungiamo i questi valori ad un array contenente tutti risultati per ogni struttura dati, per ogni tipologia di input e per ogni valore di campionamento. Infatti *finalTimesMaxTest* conterrà i valori che verranno usati per la creazione dei grafici, ovvero i risultati veri e propri.

La logica delle righe 33, 34 e 35 si ripete anche per i test del minimo, OS-select e OS-rank, anche se nello snippet 8 è stata omessa per brevità.

Andando avanti con la funzione *runAllTests*, con i risultati ottenuti andremo a creare i grafici e le tabelle. Per brevità soffermiamoci solamente sulla ricerca del massimo perché gli altri fanno riferimento alla solita logica.

```
1     #...
2     #...
3     # Reshape matrix:
4
5     # Data structures with max and min comparing are 4: max-heap, min-heap, LL
      and SLL
6     finalTimesMaxTest = finalTimesMaxTest.reshape(len(x),
      totalValueGenerationWays,totalDataStructures+1)
7
8     #...
9     #...
10
11    # Max
12    graphMaxRandValues = drawGraph("Cerca Max (valori random)", x,
      finalTimesMaxTest[:, :1, :])
13    saveGraph("max_search_with_random_values",graphMaxRandValues)
14    createAndSaveTable(x, finalTimesMaxTest[:, :1, :], '
      max_with_random_values_table')
15
16    graphMaxIncValues = drawGraph("Cerca Max (valori crescenti)", x,
      finalTimesMaxTest[:, 1:2, :])
17    saveGraph("max_search_with_increasing_values", graphMaxIncValues)
18    createAndSaveTable(x, finalTimesMaxTest[:, 1:2, :], '
      max_with_inc_values_table')
19
20    graphMaxDecValues = drawGraph("Cerca Max (valori decrescenti)", x,
      finalTimesMaxTest[:, 2:, :])
21    saveGraph("max_search_with_decreasing_values", graphMaxDecValues)
22    createAndSaveTable(x, finalTimesMaxTest[:, 2:3, :], '
      max_with_dec_values_table')
23
24    #...
25    #...
```

Frammento 11: creazione grafici e tabelle per la ricerca del massimo

Come vediamo nel frammento 11, alla riga 6 facciamo il *reshape* della matrice per trattare al meglio i dati con lo *slicing*. Dalla riga 12 alla riga 22 creiamo e salviamo i grafici relativi alle 3 tipologie di input, passando i valori relativi a quella determinata tipologia con lo *slicing*. Con la solita logica vengono passati i valori anche per creare e salvare le tabelle.

5 Analisi dei risultati

In questo paragrafo analizzeremo i risultati ottenuti in ogni test effettuato con i 3 diversi tipi di input, ovvero valori random, crescenti e decrescenti.

Piccole variazioni nei risultati (e grafici) possono essere dovute a processi in background della macchina che potrebbero influire, anche se di poco, sui risultati. Soprattutto essendo tempi molto piccoli.

Tutti i costi che vedremo saranno nei casi **medi/attesi**, in quanto sono basi appunto su una media dei tempi. In questo caso specifico saranno più vicini ai **casi peggiori** perché molte implementazioni non permettono il verificarsi del caso migliore in quanto sono basate su ricerche per confronti (max e min, forza bruta) o logiche affini.

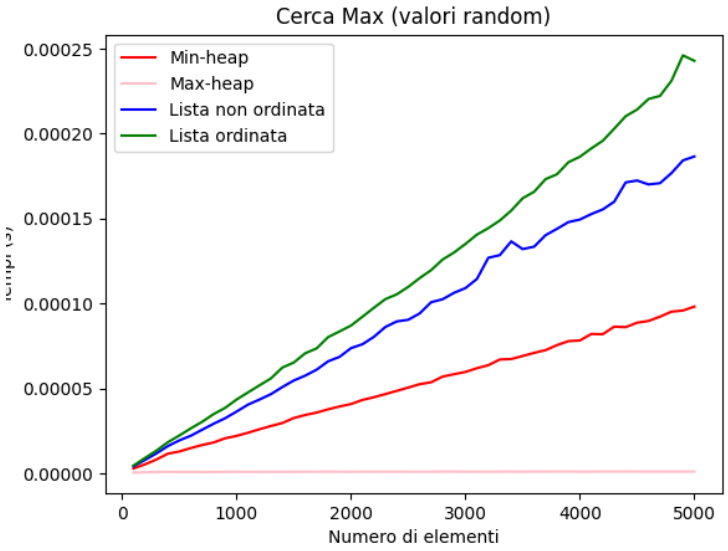
Tutte le tabelle con i risultati utilizzati per il disegno dei grafici si trova al capitolo 8.

5.1 Ricerca del massimo

Partiamo dai test riguardanti le diverse implementazioni di ricerca del massimo per le diverse strutture dati.

In questo caso sono 4 perché ci aggiungiamo anche il max-heap a scopo di dimostrare le prestazioni anche con la sua implementazione.

5.1.1 Con valori random

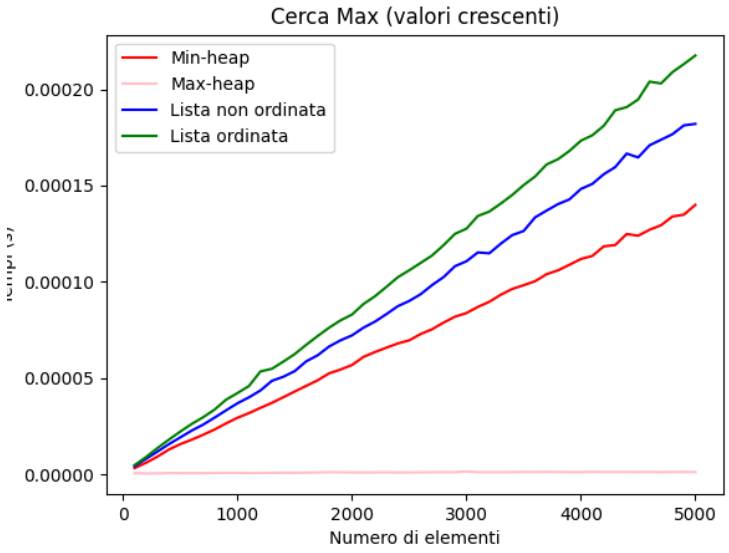


(a) Grafico

Figura 9: Grafico del test della ricerca del massimo su ogni struttura dati con valori random in input.

Come possiamo vedere dal grafico 9a, tutte le strutture dati rispettano i costi ipotizzati nella tabella 1. Infatti possiamo vedere che la ricerca del massimo con valori random in input è approssimativamente costante per il max-heap, dovendo restituire semplicemente la radice dell'heap, con un costo $O(1)$. Invece il min-heap ha un costo lineare $O(n)$, ed ha tempi minori rispetto alle altre strutture dati perché viene cercato il massimo per forza bruta sulle foglie, quindi su metà degli elementi. Le liste hanno anch'esse un costo lineare $O(n)$, in quanto ricercano il massimo per forza bruta scorrendo tutta la lista.

5.1.2 Con valori crescenti

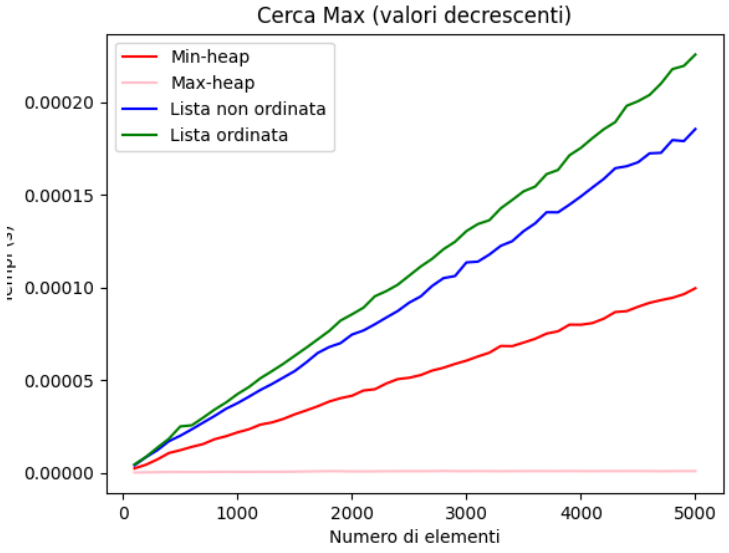


(a) Grafico

Figura 10: Grafico del test della ricerca del massimo su ogni struttura dati con valori crescenti in input.

Come possiamo vedere dal grafico 10a, tutte le strutture dati rispettano i costi ipotizzati nella tabella 1. Infatti la ricerca del massimo con valori crescenti in input ha risultati molto simili a quelli ottenuti nel paragrafo 5.1.1. Quindi valgono le solite considerazioni fatte in tale paragrafo. Abbiamo i soliti costi, ovvero $O(1)$ per il max-heap e $O(n)$ per le altre tre strutture dati nel grafico. Inoltre, notiamo che i costi non sono variati, nonostante i valori siano crescenti. Questo perché le implementazioni degli algoritmi per ogni struttura dati testata non è sensibile in modo significativo alle diverse permutazioni in input.

5.1.3 Con valori decrescenti



(a) Grafico

Figura 11: Grafico del test della ricerca del massimo su ogni struttura dati con valori decrescenti in input.

Come possiamo vedere dal grafico 11a, tutte le strutture dati rispettano i costi ipotizzati nella tabella 1. Infatti la ricerca del massimo con valori decrescenti in input ha risultati simili a quelli ottenuti nel paragrafo 5.1.1 e 5.1.2 . Quindi valgono le solite considerazioni fatte in tali paragrafi.

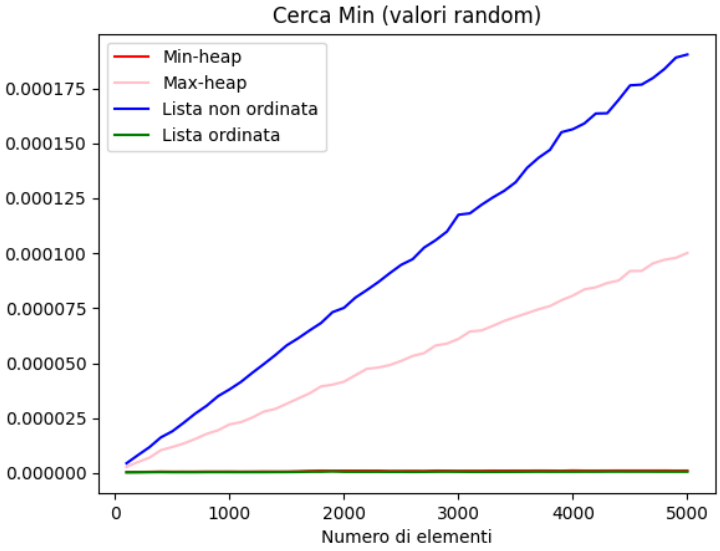
Infatti, per esempio, la lista non ordinata si comporterà come una lista ordinata (in questo caso specifico) avendo i valori in ordine crescente (essendo inseriti in testa).
Quindi avremo i soliti costi ottenuti dalle misurazioni dei paragrafi precedenti. Ovvero $O(1)$ per il max-heap e $O(n)$ per le altre tre strutture dati nel grafico.

5.2 Ricerca del minimo

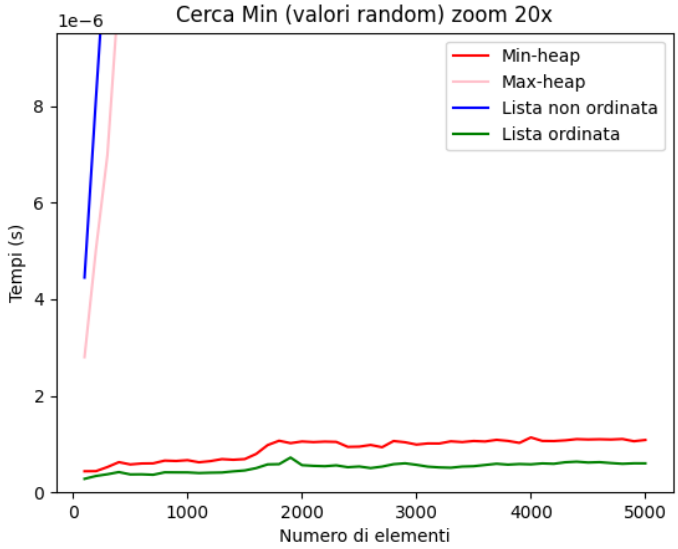
Adesso osserviamo i risultati dei test riguardanti le diverse implementazioni di ricerca del minimo per le diverse strutture dati.

Anche in questo caso sono 4 perché ci aggiungiamo anche il min-heap a scopo di dimostrare le prestazioni anche con la sua implementazione.

5.2.1 Con valori random



(a) Grafico



(b) Grafico ingrandito

Figura 12: Grafico del test della ricerca del minimo su ogni struttura dati con valori random in input.

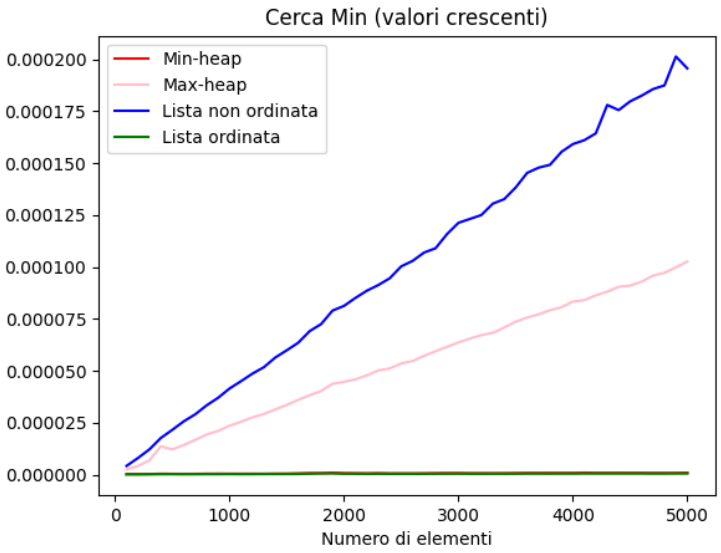
Come possiamo vedere dal grafico 12a, tutte le strutture dati rispettano i costi ipotizzati nella tabella 1.

Infatti possiamo vedere che la ricerca del minimo con valori random in input è approssimativamente costante $O(1)$ per min-heap, dovendo restituire semplicemente la radice dell'heap. Lo stesso costo vale anche per la lista ordinata, in quanto il minimo è il primo elemento della lista (la testa).

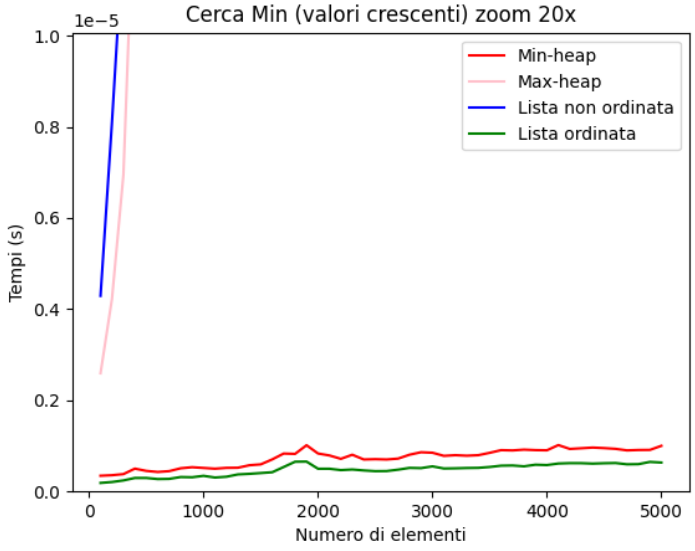
Invece il max-heap ha un costo lineare $O(n)$, ed ha tempi minori rispetto alle altre strutture dati perché viene cercato il minimo per forza bruta sulle foglie, quindi su metà degli elementi.

La lista ha anch'essa un costo lineare $O(n)$, in quanto ricercano il minimo per forza bruta scorrendo tutta la lista.

5.2.2 Con valori crescenti



(a) Grafico



(b) Grafico ingrandito

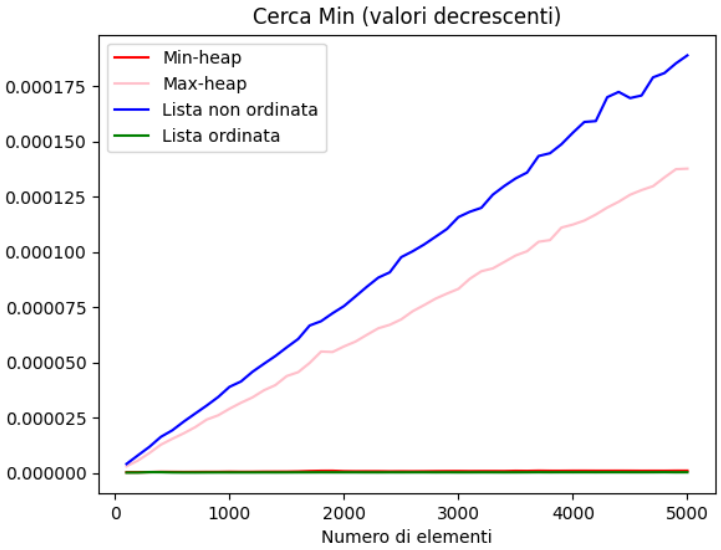
Figura 13: Grafico del test della ricerca del minimo su ogni struttura dati con valori crescenti in input.

Come vediamo dai grafici 13a e 13b, tutte le strutture dati rispettano i costi ipotizzati nella tabella 1.

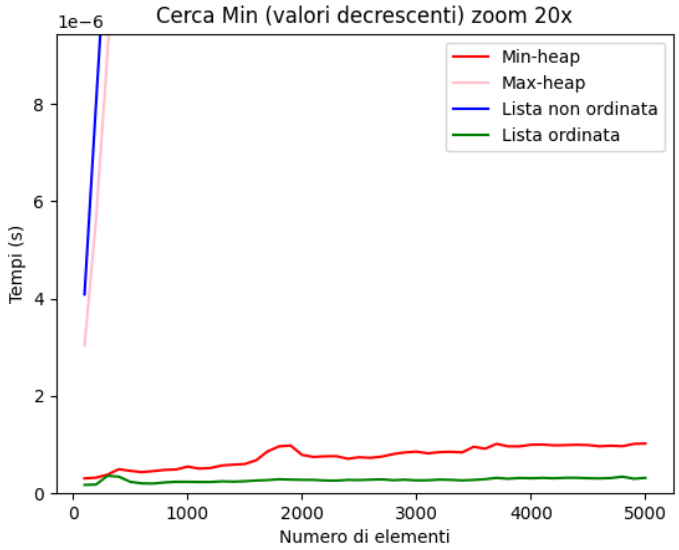
Abbiamo ottenuto risultati molto simili a quelli ottenuti nel paragrafo 5.2.1, quindi valgono le stesse considerazioni. Di conseguenza i costi saranno gli stessi, ovvero $O(1)$ per la lista ordinata e il min-heap. Invece avremo un costo di $O(n)$ la lista non ordinata e il max-heap.

Anche in questo test notiamo che i costi rimangono invariati, rispetto a quelli con valori random in input, perché le varie implementazioni di questo algoritmo sulle strutture dati esaminate non dipendono dalle permutazioni dei valori in ingresso.

5.2.3 Con valori decrescenti



(a) Grafico



(b) Grafico ingrandito

Figura 14: Grafico del test della ricerca del minimo su ogni struttura dati con valori decrescenti in input.

Come vediamo dai grafici 14a e 14b, tutte le strutture dati rispettano i costi ipotizzati nella tabella 1.

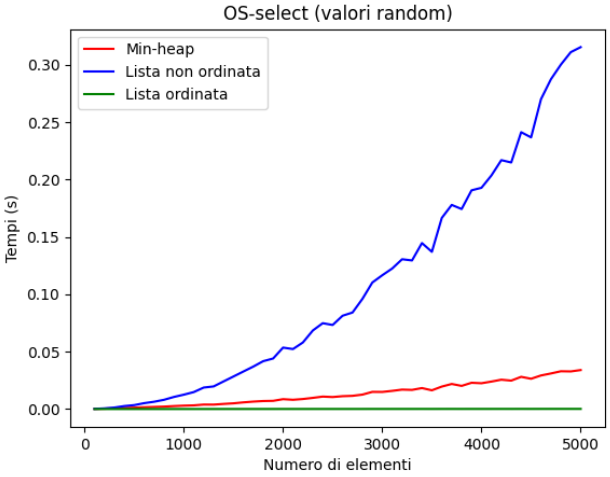
Abbiamo ottenuto risultati molto simili a quelli ottenuti nel paragrafo 5.2.1, quindi valgono le stesse considerazioni. Di conseguenza i costi saranno gli stessi, ovvero $O(1)$ per la lista ordinata e min-heap. Invece abbiamo $O(n)$ per il max-heap e la lista ordinata per i soliti motivi descritti nel paragrafo precedente.

5.3 OS-select

Adesso osserviamo i risultati dei test riguardanti le diverse implementazioni dell'algoritmo OS-select per le diverse strutture dati.

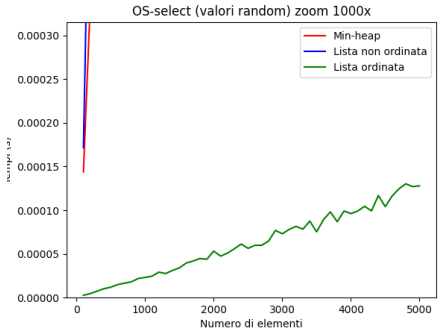
In questo caso sono 3 perché facciamo riferimento alla definizione di statistica d'ordine data nel paragrafo 2.1. Quindi, come già accennato nel paragrafo 3.1, le strutture dati in questione saranno: min-heap, lista ordinata e lista non ordinata. Nei test precedenti abbiamo usato anche l'altra tipologia di heap, solo a scopo di confronto per dimostrare l'efficienza tra i due diversi tipi di heap. In questo caso trascuriamo il max-heap.

5.3.1 Con valori random



(a) Grafico

(b) Grafico ingrandito



(c) Grafico ingrandito (1000x)

Figura 15: Grafico del test dell'algoritmo OS-select su ogni struttura dati con valori random in input.

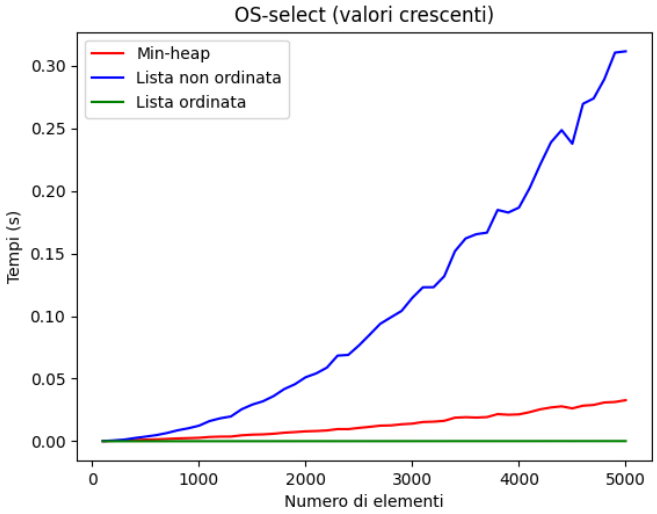
Come possiamo vedere dai grafici 15a, 15b e 15c, tutte le strutture dati rispettano i costi ipotizzati nella tabella 1.

Infatti possiamo vedere dal grafico 15b che l'OS-select con valori random in input ha un costo di $O(n \lg n)$ per min-heap, nel caso peggiore. Questo perché deve effettuare n volte l'estrazione del minimo che un costo di $O(\lg n)$ per via dell'*heapify* al suo interno. Invece il costo nel caso atteso sarebbe $O(k \lg n)$ con k estrazioni. k è il rango di un valore, ovvero la posizione del valore in un attraversamento in ordine. Nel caso migliore sarebbe $O(\lg n)$ avendo come rango 1.

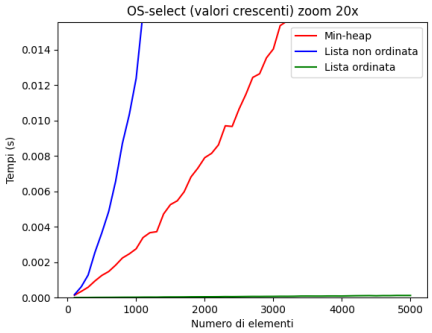
Per quanto riguarda la lista non ordinata, ha un costo di $O(n^2)$, come vediamo dal grafico 15b, infatti i tempi sono maggiori di quelli del min-heap. Questo costo è dovuto all'implementazione dell'algoritmo in questa struttura dati, perché essa è basata sul Selection-sort che ha un costo quadratico in ogni caso. Come descritto nel paragrafo 2.2.2.

La lista ordinata, come vediamo del grafico 15c, ha un costo lineare $O(n)$. Infatti il tempo cresce linearmente all'aumentare della dimensione della lista. Tenendo conto che i valori k in input vengono generati casualmente ad ogni iterazione per aumentare l'affidabilità dei risultati.

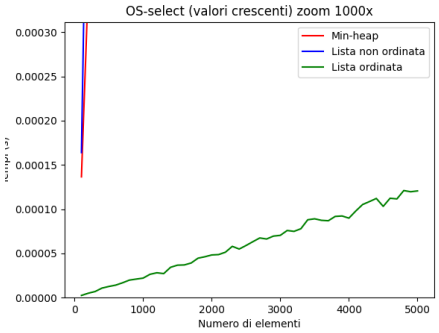
5.3.2 Con valori crescenti



(a) Grafico



(b) Grafico ingrandito



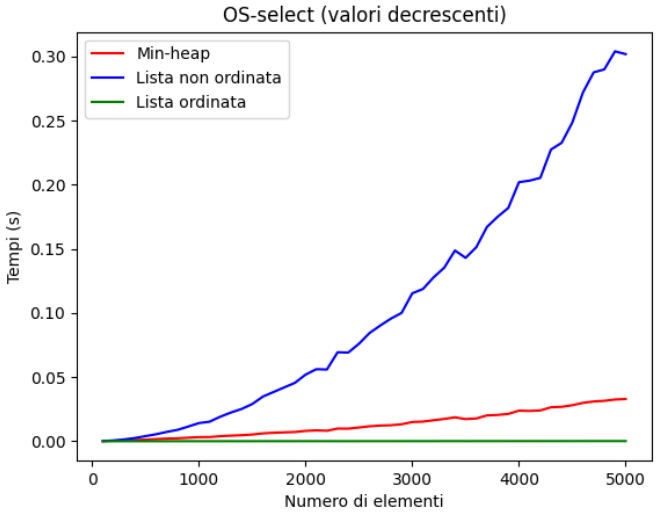
(c) Grafico ingrandito (1000x)

Figura 16: Grafico del test dell'algoritmo OS-select su ogni struttura dati con valori crescenti in input.

Come possiamo vedere dai grafici 16a, 16b e 16c, tutte le strutture dati rispettano i costi ipotizzati nella tabella 1.

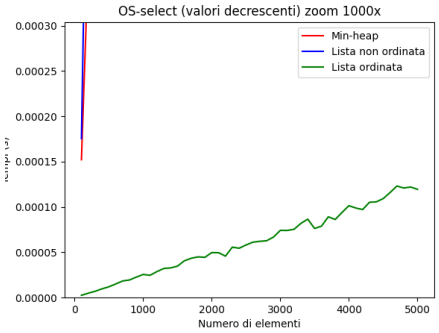
Le considerazioni fatte sono analoghe a quelle del paragrafo 5.3.1, perché in questo caso i valori in input non fanno variare drasticamente i costi di esecuzione. In particolar modo, nell'heap non è rilevante la permutazione di ingresso in quanto i valori vengono collocati al giusto posto dall'*heapify*. Infatti il costo rimane di $O(n \lg n)$. Lo stesso vale per la lista ordinata, perché viene ordinata all'inserimento rendendo invariato il costo $O(n)$ indipendentemente dalla permutazione in input o dalle diverse disposizioni dei valori.
Invece, per la lista non ordinata abbiamo il solito costo $O(n^2)$ perché l'implementazione è basata su Selection-sort, il quale non tiene conto della permutazione in ingresso. Maggiori dettagli sono dati nei paragrafi 5.3.1 e 2.1.

5.3.3 Con valori decrescenti



(a) Grafico

(b) Grafico ingrandito



(c) Grafico ingrandito (1000x)

Figura 17: Grafico del test dell'algoritmo OS-select su ogni struttura dati con valori decrescenti in input.

Come possiamo vedere dai grafici 17a, 17b e 17c, tutte le strutture dati rispettano i costi ipotizzati nella tabella 1.

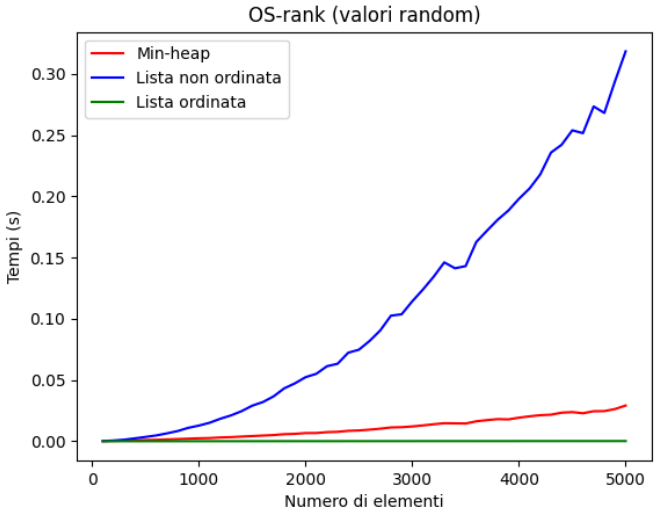
Valgono le considerazioni fatte nel paragrafo 5.3.2 con i soliti costi derivanti dai risultati ottenuti molto simili. Infatti abbiamo nuovamente un costo di $O(n \lg n)$ per il min-heap, $O(n^2)$ per la lista non ordinata e $O(n)$ per la lista ordinata.

5.4 OS-rank

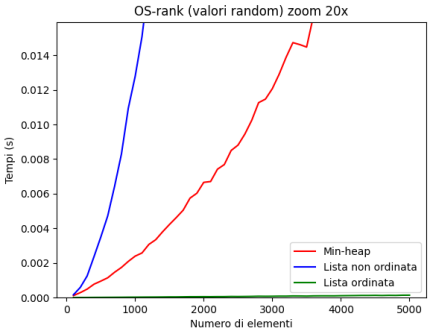
Adesso osserviamo i risultati dei test riguardanti le diverse implementazioni dell'algoritmo OS-rank per le diverse strutture dati.

Anche in questo caso sono 3, come spiegato nel paragrafo 5.3.1, perché facciamo riferimento alla definizione di statistica d'ordine data nel paragrafo 2.1. Quindi le strutture dati saranno: min-heap, lista ordinata e lista non ordinata.

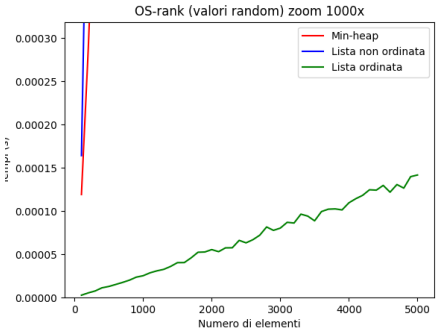
5.4.1 Con valori random



(a) Grafico



(b) Grafico ingrandito



(c) Grafico ingrandito (1000x)

Figura 18: Grafico del test dell'algoritmo OS-rank su ogni struttura dati con valori random in input.

Come possiamo vedere dai grafici 18a, 18b e 18c, tutte le strutture dati rispettano i costi ipotizzati nella tabella 1.

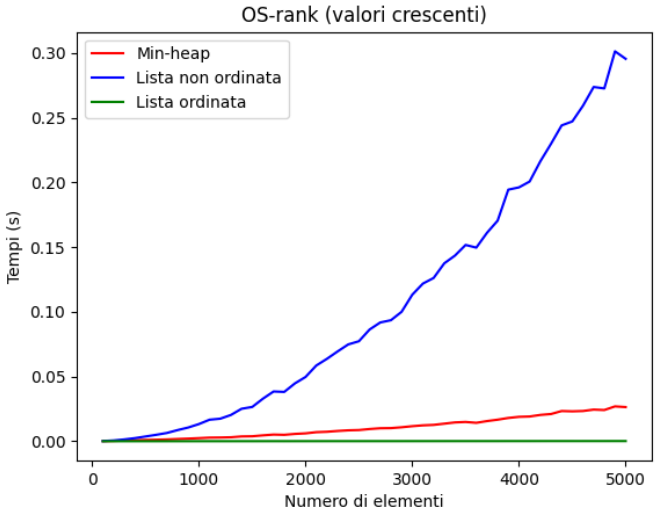
Infatti possiamo vedere dal grafico 18b che abbiamo ottenuto risultati molto simili a quelli del test al paragrafo 5.3.1. Di conseguenza avremo costi analoghi.

Quindi con valori random in input l'OS-rank ha un costo di $O(n \lg n)$ per min-heap, nel caso peggiore. Questo perché deve effettuare n volte l'estrazione del minimo che ha un costo di $O(\lg n)$ per via dell'*heapify* al suo interno. Il numero di estrazioni sarà contato in modo tale da restituire il rango quando viene trovato il valore di input. Nel caso migliore sarebbe $O(\lg n)$ avendo la radice dell'min-heap come primo valore più piccolo, ovvero come rango 1.

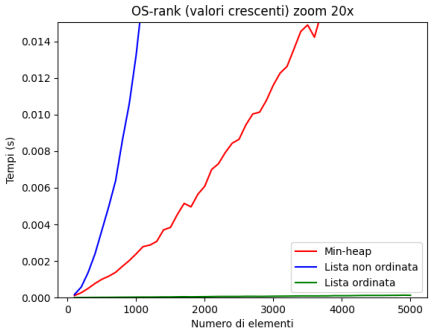
Per quanto riguarda la lista non ordinata che ha un costo di $O(n^2)$, come vediamo dal grafico 18b. Infatti i tempi sono maggiori di quelli del min-heap. Questo costo è dovuto all'implementazione dell'algoritmo in questa struttura dati, perché essa è basata sul Selection-sort (come OS-select) che ha un costo quadratico in ogni caso. Come descritto nel paragrafo 2.2.2.

Anche qui la lista ordinata, come vediamo del grafico 18c, ha un costo lineare $O(n)$. Infatti il tempo cresce linearmente all'aumentare della dimensione della lista. Praticamente viene restituito il valore del contatore che viene incrementato man mano che si scorre la lista. Il valore viene restituito quando viene trovato il valore dato in input (come una ricerca) oppure quando la lista è finita. Quindi il tempo è analogo ad una ricerca per forza bruta in una lista ordinata.

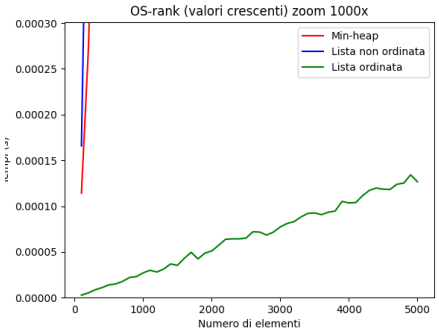
5.4.2 Con valori crescenti



(a) Grafico



(b) Grafico ingrandito



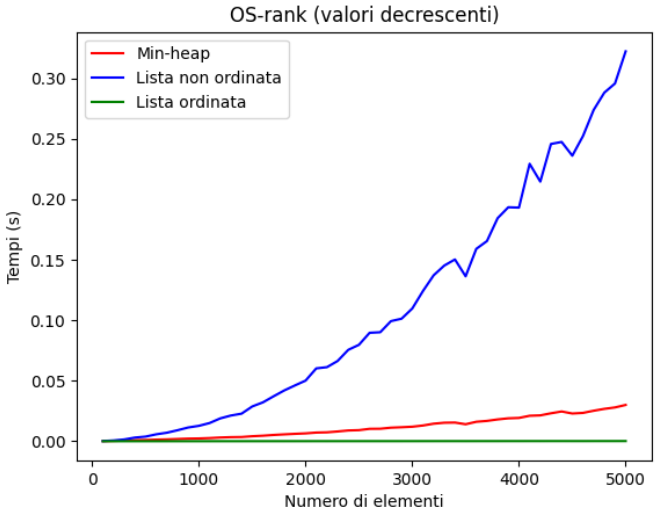
(c) Grafico ingrandito (1000x)

Figura 19: Grafico del test dell'algoritmo OS-rank su ogni struttura dati con valori crescenti in input.

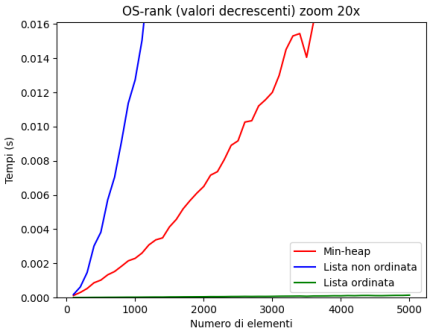
Come possiamo vedere dai grafici 19a, 19b e 19c, tutte le strutture dati rispettano i costi ipotizzati nella tabella 1.

Anche in questo caso, i risultati ottenuti sono molto simili a quelli della paragrafo 5.4.1. Di conseguenza i costi saranno analoghi e valgono anche le stesse considerazioni, fatte anche al paragrafo 5.3.2 per l'OS-select.

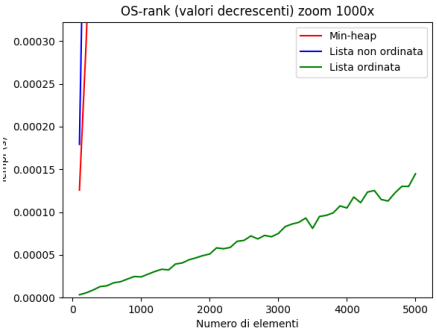
5.4.3 Con valori decrescenti



(a) Grafico



(b) Grafico ingrandito



(c) Grafico ingrandito (1000x)

Figura 20: Grafico del test dell'algoritmo OS-rank su ogni struttura dati con valori decrescenti in input.

Come possiamo vedere dai grafici 20a, 20b e 20c, tutte le strutture dati rispettano i costi ipotizzati nella tabella 1.

Anche in questo caso, i risultati ottenuti sono molto simili a quelli dei paragrafi precedenti. Di conseguenza i costi saranno analoghi e valgono anche le stesse considerazioni, fatte anche al paragrafo 5.3.2 per l'OS-select.

6 Conclusioni

Abbiamo visto i risultati ed i grafici di questi 4 esperimenti, di cui ognuno di essi è stato testato con 3 diverse tipologie di input.

Abbiamo potuto osservare che tutti gli esperimenti rispettano i costi previsti nella tabella 1.

In particolare abbiamo visto che i risultati ottenuti con le diverse implementazioni di **OS-select** e **OS-rank** sulle diverse strutture dati portano più o meno gli stessi risultati con ogni tipologia di input testata, quindi non nono sensibili alle diverse tipologie di input. Infatti possiamo dire che una lista ordinata potrebbe essere la struttura dati più efficiente (tra quelle testate) per eseguire questi due algoritmi in quanto ha un costo di $O(n)$. Invece, la lista non ordinata è quella più inefficiente, anche intuitivamente, perché in qualche modo dovremo ordinare i valori e tale ordinamento avrà un costo.

Passando alla **ricerca del minimo** abbiamo visto, come da ipotesi, che la struttura dati max-heap è inefficiente rispetto al min-heap, in quanto quest'ultimo ha caratteristiche più adatte. Infatti quest'ultima ha un costo costante $O(1)$, così come la lista ordinata (in modo crescente) perché ha il minimo nella testa. D'altro canto, una lista non ordinata è meno efficiente di un max-heap per la ricerca del minimo. Questo perché il max-heap ha la caratteristica di contenere il minimo nelle foglie, ovvero nella seconda metà dell'array dove è memorizzato l'heap. In questo modo la ricerca verrà effettuata su $\lceil \frac{n}{2} \rceil$ elementi. A differenza della lista non ordinata che deve effettuare una vera e propria ricerca per forza bruta, scorrendo sistematicamente tutta la lista per confronti, quindi su n elementi.

Infine, per quanto riguarda la **ricerca del massimo**, abbiamo potuto osservare che la struttura dati (tra quelle testate) più efficiente è il max-heap con costo costante $O(1)$. Anche se inefficiente in confronto al min-heap, analogamente a quanto detto prima, il max-heap ha il minimo nella seconda metà dell'array, quindi effettua la ricerca del minimo per forza bruta su $\lceil \frac{n}{2} \rceil$ elementi. Invece le liste (ordinata e non) hanno il solito costo lineare perché la ricerca del massimo avviene per forza bruta su n elementi, scorrendo tutta la lista sistematicamente.

In conclusione, notiamo che ognuno dei 4 test effettuati, anche se effettuato 3 diverse tipologie di valori, presentano risultati molto simili (per ogni test). Questo perché gli algoritmi che vengono testati non dipendono dalle diverse permutazioni in input. Il costo principale dipende dalle proprietà delle srtutture dati implementate per quanto riguarda gli heap. Invece per quanto riguarda le liste non ordinate dobbiamo in ogni caso scorrere la lista per trovare le statistiche d'ordine (essendo non ordinata). Invece per la lista ordinata varia a seconda dei casi, alcune volte possiamo usare le proprietà della struttura dati ed in alcune volte è necessario

7 Risorse usate

- Slide del corso di Algoritmi e Strutture Dati
- Libro del corso: Introduzione agli algoritmi e strutture dati di Cromen, Leiserson, Rivest e Stein. Editore Mc-Graw Hill.
- https://www.geeksforgeeks.org/data-structures/linked-list/
- https://afteracademy.com/blog/introduction-to-heaps-in-data-structures/
- https://www.matematicamente.it/forum/viewtopic.php?f=15&t=186082&start=10

7.1 Strumenti usati

- LucidChart Online: per la creazione dei diagrammi UML

8 Tabelle dei risultati

8.1 Ricerca del massimo

8.1.1 Valori random

	Min-Heap	Lista non ordinata	Lista ordinata	Max-Heap
100	2.94934e-06	4.20066e-06	4.65133e-06	5.33333e-07
200	5.34999e-06	7.94732e-06	9.12267e-06	7.09345e-07
300	8.23932e-06	1.18333e-05	1.34413e-05	8.39336e-07
400	1.1638e-05	1.61240e-05	1.8408e-05	9.60672e-07
500	1.29507e-05	1.94707e-05	2.23407e-05	8.94677e-07
600	1.49533e-05	2.22007e-05	2.65373e-05	8.95330e-07
700	1.67633e-05	2.57613e-05	3.04447e-05	8.68657e-07
800	1.82633e-05	2.92847e-05	3.49693e-05	8.76654e-07
900	2.07067e-05	3.25033e-05	3.86007e-05	9.90671e-07
1000	2.2112e-05	3.6422e-05	4.35140e-05	9.48656e-07
1100	2.398e-05	4.05327e-05	4.76907e-05	9.16001e-07
1200	2.60233e-05	4.35007e-05	5.18627e-05	9.22664e-07
1300	2.79427e-05	4.66660e-05	5.59433e-05	9.68662e-07
1400	2.96940e-05	5.08567e-05	6.23293e-05	9.88004e-07
1500	3.26313e-05	5.47113e-05	6.52987e-05	9.58003e-07
1600	3.44260e-05	5.75933e-05	7.07627e-05	9.95323e-07
1700	3.58407e-05	6.11147e-05	7.36313e-05	1.02133e-06
1800	3.77660e-05	6.60153e-05	8.02707e-05	1.11599e-06
1900	3.94380e-05	6.86387e-05	8.36127e-05	1.02667e-06
2000	4.08707e-05	7.37740e-05	8.706e-05	9.98679e-07
2100	4.32827e-05	7.60567e-05	9.22567e-05	1.03999e-06
2200	4.48767e-05	8.04047e-05	9.75273e-05	1.03866e-06
2300	4.67287e-05	8.62147e-05	1.02603e-04	1.05600e-06
2400	4.86133e-05	8.9494e-05	1.05450e-04	1.04933e-06
2500	5.05580e-05	9.0466e-05	1.09821e-04	1.03801e-06
2600	5.25700e-05	9.42047e-05	1.15009e-04	1.03667e-06
2700	5.36867e-05	1.00811e-04	1.19674e-04	1.02334e-06
2800	5.70073e-05	1.02527e-04	1.25901e-04	1.11667e-06
2900	5.84207e-05	1.06317e-04	1.30063e-04	1.08934e-06
3000	5.97633e-05	1.09126e-04	1.35091e-04	1.032e-06
3100	6.1968e-05	1.14467e-04	1.40569e-04	1.05800e-06
3200	6.37413e-05	1.26963e-04	1.44421e-04	1.068e-06
3300	6.70867e-05	1.28492e-04	1.48873e-04	1.08734e-06
3400	6.73533e-05	1.36613e-04	1.54731e-04	1.10665e-06
3500	6.91553e-05	1.32061e-04	1.61993e-04	1.05865e-06
3600	7.09827e-05	1.33487e-04	1.65809e-04	1.118e-06
3700	7.25947e-05	1.40257e-04	1.73231e-04	1.08533e-06
3800	7.55313e-05	1.43989e-04	1.76129e-04	1.15200e-06
3900	7.79014e-05	1.47973e-04	1.83215e-04	1.16134e-06
4000	7.82753e-05	1.49461e-04	1.86431e-04	1.05533e-06
4100	8.21053e-05	1.52703e-04	1.91331e-04	1.08801e-06
4200	8.19767e-05	1.55415e-04	1.95832e-04	1.15734e-06
4300	8.64193e-05	1.5997e-04	2.02941e-04	1.12268e-06
4400	8.61240e-05	1.71327e-04	2.10287e-04	1.17001e-06
4500	8.875e-05	1.72427e-04	2.14210e-04	1.12600e-06
4600	8.97887e-05	1.70121e-04	2.20425e-04	1.10199e-06
4700	9.23113e-05	1.70879e-04	2.2223e-04	1.11067e-06
4800	9.52440e-05	1.76795e-04	2.31159e-04	1.09334e-06
4900	9.59047e-05	1.84239e-04	2.46089e-04	1.09600e-06
5000	9.82207e-05	1.86595e-04	2.42885e-04	1.102e-06

Tabella 2: Tabella risultati della ricerca del massimo con valori random

8.1.2 Valori crescenti

	Min-Heap	Lista non ordinata	Lista ordinata	Max-Heap
100	3.24468e-06	4.21132e-06	4.74467e-06	5.76678e-07
200	5.94799e-06	7.99134e-06	8.94666e-06	4.46679e-07
300	9.17399e-06	1.17953e-05	1.36187e-05	4.92002e-07
400	1.28407e-05	1.56373e-05	1.79773e-05	6.57331e-07
500	1.56667e-05	1.91787e-05	2.2206e-05	6.18002e-07
600	1.79893e-05	2.27180e-05	2.61753e-05	6.06662e-07
700	2.05467e-05	2.58073e-05	2.966e-05	6.10668e-07
800	2.32913e-05	2.94667e-05	3.36373e-05	6.70670e-07
900	2.6438e-05	3.32300e-05	3.877e-05	7.26003e-07
1000	2.93827e-05	3.69080e-05	4.221e-05	7.16009e-07
1100	3.17820e-05	3.9998e-05	4.58767e-05	6.38004e-07
1200	3.44947e-05	4.35413e-05	5.34227e-05	7.01336e-07
1300	3.7116e-05	4.85473e-05	5.4806e-05	7.50654e-07
1400	4.00453e-05	5.06593e-05	5.85347e-05	7.84003e-07
1500	4.30087e-05	5.36133e-05	6.24633e-05	7.77336e-07
1600	4.5926e-05	5.86753e-05	6.72573e-05	8.64003e-07
1700	4.88253e-05	6.18220e-05	7.18226e-05	9.74654e-07
1800	5.24447e-05	6.63727e-05	7.61573e-05	1.07333e-06
1900	5.44280e-05	6.95947e-05	7.9966e-05	1.03999e-06
2000	5.67693e-05	7.21573e-05	8.298e-05	9.23998e-07
2100	6.09707e-05	7.61587e-05	8.84413e-05	9.22667e-07
2200	6.34907e-05	7.92853e-05	9.24560e-05	9.55332e-07
2300	6.57653e-05	8.31733e-05	9.73273e-05	1.02332e-06
2400	6.79673e-05	8.73167e-05	1.02372e-04	8.96668e-07
2500	6.9584e-05	9.0042e-05	1.05981e-04	9.48670e-07
2600	7.28600e-05	9.35507e-05	1.09811e-04	9.93347e-07
2700	7.53667e-05	9.83107e-05	1.13691e-04	1.05467e-06
2800	7.87887e-05	1.02421e-04	1.19021e-04	1.11666e-06
2900	8.18747e-05	1.08117e-04	1.2491e-04	1.05266e-06
3000	8.37613e-05	1.10673e-04	1.27599e-04	1.51600e-06
3100	8.69640e-05	1.15269e-04	1.34203e-04	1.06468e-06
3200	8.96280e-05	1.14812e-04	1.36494e-04	1.07867e-06
3300	9.32840e-05	1.19799e-04	1.40639e-04	1.08867e-06
3400	9.62987e-05	1.24245e-04	1.45109e-04	1.11267e-06
3500	9.8244e-05	1.26394e-04	1.50251e-04	1.19132e-06
3600	1.00292e-04	1.33481e-04	1.54701e-04	1.13268e-06
3700	1.03953e-04	1.36981e-04	1.60939e-04	1.23134e-06
3800	1.05953e-04	1.40351e-04	1.63695e-04	1.15468e-06
3900	1.08854e-04	1.42813e-04	1.67985e-04	1.12600e-06
4000	1.11857e-04	1.48234e-04	1.73309e-04	1.11667e-06
4100	1.13429e-04	1.50857e-04	1.76115e-04	1.24534e-06
4200	1.18418e-04	1.55894e-04	1.81053e-04	1.17e-06
4300	1.19095e-04	1.59577e-04	1.89091e-04	1.14666e-06
4400	1.24835e-04	1.66623e-04	1.9074e-04	1.17134e-06
4500	1.23942e-04	1.64593e-04	1.94707e-04	1.12734e-06
4600	1.27039e-04	1.70919e-04	2.03991e-04	1.15866e-06
4700	1.29371e-04	1.73788e-04	2.03036e-04	1.11667e-06
4800	1.33911e-04	1.76673e-04	2.08907e-04	1.15266e-06
4900	1.34881e-04	1.81263e-04	2.13128e-04	1.21267e-06
5000	1.39963e-04	1.82038e-04	2.17543e-04	1.124e-06

Tabella 3: Tabella risultati della ricerca del massimo con valori crescenti

8.1.3 Valori decrescenti

	Min-Heap	Lista non ordinata	Lista ordinata	Max-Heap
100	2.508e-06	4.20334e-06	4.65800e-06	3.18003e-07
200	4.47999e-06	8.55e-06	8.72200e-06	3.77993e-07
300	7.34732e-06	1.23453e-05	1.37253e-05	4.72674e-07
400	1.07887e-05	1.71513e-05	1.84773e-05	5.78666e-07
500	1.23207e-05	2.01367e-05	2.51407e-05	5.5001e-07
600	1.40960e-05	2.3552e-05	2.5732e-05	5.53991e-07
700	1.56427e-05	2.7214e-05	2.98873e-05	5.51326e-07
800	1.82473e-05	3.08060e-05	3.42400e-05	6.26664e-07
900	1.9868e-05	3.46873e-05	3.80747e-05	6.39996e-07
1000	2.19260e-05	3.77087e-05	4.2614e-05	6.25996e-07
1100	2.36180e-05	4.11987e-05	4.641e-05	6.21333e-07
1200	2.6116e-05	4.48120e-05	5.11180e-05	6.54653e-07
1300	2.72780e-05	4.79993e-05	5.49327e-05	6.4134e-07
1400	2.91813e-05	5.14993e-05	5.8918e-05	6.56000e-07
1500	3.1664e-05	5.5012e-05	6.32513e-05	7.42666e-07
1600	3.37653e-05	5.9656e-05	6.75693e-05	8.52008e-07
1700	3.59860e-05	6.4738e-05	7.208e-05	9.30016e-07
1800	3.85473e-05	6.79453e-05	7.66047e-05	1.02866e-06
1900	4.03753e-05	7.00813e-05	8.22327e-05	9.79326e-07
2000	4.16887e-05	7.46887e-05	8.56047e-05	8.78653e-07
2100	4.45027e-05	7.68953e-05	8.91887e-05	8.80662e-07
2200	4.5208e-05	8.02433e-05	9.53507e-05	9.03999e-07
2300	4.82967e-05	8.38827e-05	9.8138e-05	9.70014e-07
2400	5.0668e-05	8.74133e-05	1.01593e-04	9.87995e-07
2500	5.13873e-05	9.19207e-05	1.06491e-04	1.00333e-06
2600	5.28293e-05	9.52540e-05	1.11377e-04	1.00865e-06
2700	5.52493e-05	1.00903e-04	1.15551e-04	1.01133e-06
2800	5.68153e-05	1.05155e-04	1.20757e-04	1.11200e-06
2900	5.88433e-05	1.06255e-04	1.24731e-04	1.01866e-06
3000	6.06327e-05	1.13612e-04	1.30515e-04	9.92009e-07
3100	6.2862e-05	1.14031e-04	1.34303e-04	1.02799e-06
3200	6.49047e-05	1.17849e-04	1.36367e-04	1.012e-06
3300	6.85513e-05	1.22555e-04	1.42783e-04	9.79998e-07
3400	6.845e-05	1.25069e-04	1.47302e-04	1.01799e-06
3500	7.03893e-05	1.30503e-04	1.51977e-04	1.018e-06
3600	7.23960e-05	1.34657e-04	1.54504e-04	1.04200e-06
3700	7.52033e-05	1.40677e-04	1.61217e-04	1.02735e-06
3800	7.64600e-05	1.40635e-04	1.63492e-04	1.01733e-06
3900	8.00380e-05	1.44767e-04	1.71389e-04	1.08134e-06
4000	8.00027e-05	1.49208e-04	1.75400e-04	1.02199e-06
4100	8.08807e-05	1.53989e-04	1.80611e-04	1.00999e-06
4200	8.32753e-05	1.58656e-04	1.85450e-04	1.02999e-06
4300	8.68487e-05	1.64426e-04	1.89305e-04	1.03734e-06
4400	8.73220e-05	1.65494e-04	1.98085e-04	1.06000e-06
4500	8.97147e-05	1.67664e-04	2.00593e-04	1.05666e-06
4600	9.18093e-05	1.72453e-04	2.03963e-04	1.02467e-06
4700	9.32673e-05	1.72738e-04	2.10117e-04	9.73996e-07
4800	9.45373e-05	1.79613e-04	2.17896e-04	1.02134e-06
4900	9.64920e-05	1.79061e-04	2.19649e-04	1.03665e-06
5000	9.9678e-05	1.85561e-04	2.25758e-04	1.04466e-06

Tabella 4: Tabella risultati della ricerca del massimo con valori decrescenti

8.2 Ricerca del minimo

8.2.1 Valori random

	Min-Heap	Lista non ordinata	Lista ordinata	Max-Heap
100	4.36002e-07	4.446e-06	2.79327e-07	2.79867e-06
200	4.37999e-07	8.16134e-06	3.38668e-07	5.02934e-06
300	5.23333e-07	1.17580e-05	3.73994e-07	6.984e-06
400	6.24661e-07	1.62133e-05	4.1933e-07	1.03993e-05
500	5.77338e-07	1.898e-05	3.70663e-07	1.18380e-05
600	5.97347e-07	2.28933e-05	3.72664e-07	1.34507e-05
700	6.00006e-07	2.70313e-05	3.63338e-07	1.54873e-05
800	6.56665e-07	3.06107e-05	4.13999e-07	1.78053e-05
900	6.4866e-07	3.49780e-05	4.12675e-07	1.946e-05
1000	6.65326e-07	3.80373e-05	4.12000e-07	2.21220e-05
1100	6.22012e-07	4.1464e-05	3.99332e-07	2.31547e-05
1200	6.46004e-07	4.562e-05	4.05995e-07	2.52747e-05
1300	6.87335e-07	4.95800e-05	4.10007e-07	2.79433e-05
1400	6.73995e-07	5.36553e-05	4.34009e-07	2.917e-05
1500	6.88002e-07	5.80827e-05	4.52680e-07	3.15240e-05
1600	7.95329e-07	6.13067e-05	5.01337e-07	3.3918e-05
1700	9.77999e-07	6.48880e-05	5.78004e-07	3.62713e-05
1800	1.068e-06	6.82600e-05	5.83342e-07	3.93987e-05
1900	1.01801e-06	7.31793e-05	7.19995e-07	4.021e-05
2000	1.05268e-06	7.51307e-05	5.61996e-07	4.15560e-05
2100	1.04000e-06	7.98e-05	5.47989e-07	4.44953e-05
2200	1.05001e-06	8.32307e-05	5.40678e-07	4.73993e-05
2300	1.044e-06	8.6856e-05	5.58008e-07	4.79860e-05
2400	9.4133e-07	9.08973e-05	5.19995e-07	4.91047e-05
2500	9.45997e-07	9.4696e-05	5.34671e-07	5.10040e-05
2600	9.80667e-07	9.727e-05	5.01991e-07	5.32440e-05
2700	9.33991e-07	1.02529e-04	5.32659e-07	5.45540e-05
2800	1.06200e-06	1.0586e-04	5.81997e-07	5.79847e-05
2900	1.03667e-06	1.09849e-04	6.00666e-07	5.88993e-05
3000	9.88656e-07	1.17483e-04	5.69335e-07	6.09727e-05
3100	1.01199e-06	1.18105e-04	5.31335e-07	6.43693e-05
3200	1.01067e-06	1.21936e-04	5.15999e-07	6.48267e-05
3300	1.05600e-06	1.25319e-04	5.09336e-07	6.69107e-05
3400	1.04066e-06	1.2835e-04	5.33331e-07	6.91787e-05
3500	1.06268e-06	1.32266e-04	5.40678e-07	7.09787e-05
3600	1.05266e-06	1.38819e-04	5.65991e-07	7.27127e-05
3700	1.08668e-06	1.43379e-04	5.91328e-07	7.44760e-05
3800	1.064e-06	1.47003e-04	5.73339e-07	7.59487e-05
3900	1.02398e-06	1.55004e-04	5.86001e-07	7.85927e-05
4000	1.13599e-06	1.56383e-04	5.79329e-07	8.06793e-05
4100	1.06399e-06	1.58989e-04	5.98674e-07	8.3544e-05
4200	1.06133e-06	1.63503e-04	5.90657e-07	8.44773e-05
4300	1.07601e-06	1.6364e-04	6.22661e-07	8.64120e-05
4400	1.10133e-06	1.69763e-04	6.34659e-07	8.75827e-05
4500	1.09399e-06	1.76323e-04	6.17324e-07	9.1892e-05
4600	1.09868e-06	1.76644e-04	6.24655e-07	9.1924e-05
4700	1.09332e-06	1.79675e-04	6.05999e-07	9.5356e-05
4800	1.10401e-06	1.83774e-04	5.90667e-07	9.7008e-05
4900	1.05733e-06	1.88959e-04	6.00667e-07	9.787e-05
5000	1.08400e-06	1.90338e-04	6.00001e-07	1.00097e-04

Tabella 5: Tabella risultati della ricerca del minimo con valori random

8.2.2 Valori crescenti

	Min-Heap	Lista non ordinata	Lista ordinata	Max-Heap
100	3.44658e-07	4.29133e-06	1.86667e-07	2.59333e-06
200	3.55990e-07	8.10269e-06	2.06667e-07	4.22466e-06
300	3.79994e-07	1.22447e-05	2.4134e-07	6.94867e-06
400	5.00657e-07	1.775e-05	2.95322e-07	1.37260e-05
500	4.50669e-07	2.16520e-05	2.95346e-07	1.22627e-05
600	4.2733e-07	2.5736e-05	2.71332e-07	1.43287e-05
700	4.4534e-07	2.91213e-05	2.75324e-07	1.68300e-05
800	5.08012e-07	3.34487e-05	3.15993e-07	1.94193e-05
900	5.31324e-07	3.7076e-05	3.11326e-07	2.11320e-05
1000	5.15344e-07	4.15500e-05	3.42671e-07	2.36186e-05
1100	4.99324e-07	4.50587e-05	3.05342e-07	2.5532e-05
1200	5.1735e-07	4.87187e-05	3.22663e-07	2.76193e-05
1300	5.19992e-07	5.18140e-05	3.73332e-07	2.92707e-05
1400	5.74666e-07	5.64853e-05	3.86004e-07	3.14900e-05
1500	5.93982e-07	5.99540e-05	4.03336e-07	3.36387e-05
1600	7.02666e-07	6.35847e-05	4.21327e-07	3.60960e-05
1700	8.3133e-07	6.91787e-05	5.35990e-07	3.83020e-05
1800	8.24011e-07	7.25653e-05	6.52000e-07	4.03813e-05
1900	1.01399e-06	7.90773e-05	6.54655e-07	4.37893e-05
2000	8.32664e-07	8.13247e-05	4.98001e-07	4.47627e-05
2100	7.89342e-07	8.51767e-05	4.97995e-07	4.5968e-05
2200	7.15347e-07	8.86447e-05	4.68000e-07	4.78793e-05
2300	8.05335e-07	9.13687e-05	4.81331e-07	5.02133e-05
2400	7.01329e-07	9.46480e-05	4.60654e-07	5.1222e-05
2500	7.0799e-07	1.00366e-04	4.45332e-07	5.36253e-05
2600	7.01331e-07	1.03013e-04	4.47333e-07	5.48173e-05
2700	7.20664e-07	1.0702e-04	4.78003e-07	5.72760e-05
2800	8.06004e-07	1.09076e-04	5.16673e-07	5.9492e-05
2900	8.59997e-07	1.15896e-04	5.11319e-07	6.16500e-05
3000	8.49348e-07	1.21336e-04	5.50007e-07	6.37207e-05
3100	7.80667e-07	1.23191e-04	5.03344e-07	6.55887e-05
3200	7.95336e-07	1.25071e-04	5.07343e-07	6.72833e-05
3300	7.84001e-07	1.30552e-04	5.13997e-07	6.83393e-05
3400	7.95326e-07	1.32735e-04	5.1733e-07	7.09073e-05
3500	8.47336e-07	1.38342e-04	5.3667e-07	7.37373e-05
3600	9.05994e-07	1.45357e-04	5.65321e-07	7.56867e-05
3700	8.99999e-07	1.47879e-04	5.69347e-07	7.72887e-05
3800	9.17322e-07	1.49263e-04	5.51326e-07	7.92747e-05
3900	9.06666e-07	1.55481e-04	5.86659e-07	8.06600e-05
4000	9.02663e-07	1.5924e-04	5.78665e-07	8.34233e-05
4100	1.01732e-06	1.61122e-04	6.12669e-07	8.40893e-05
4200	9.32665e-07	1.64393e-04	6.2068e-07	8.6412e-05
4300	9.47337e-07	1.78089e-04	6.20003e-07	8.814e-05
4400	9.62665e-07	1.75599e-04	6.10664e-07	9.05273e-05
4500	9.50669e-07	1.79795e-04	6.18674e-07	9.11173e-05
4600	9.34675e-07	1.82541e-04	6.23334e-07	9.29647e-05
4700	9.00671e-07	1.85708e-04	5.95322e-07	9.58947e-05
4800	9.09334e-07	1.87497e-04	5.98665e-07	9.72720e-05
4900	9.11341e-07	2.01397e-04	6.47997e-07	9.98893e-05
5000	1.00066e-06	1.95675e-04	6.33999e-07	1.02675e-04

Tabella 6: Tabella risultati della ricerca del minimo con valori crescenti

8.2.3 Valori decrescenti

	Min-Heap	Lista non ordinata	Lista ordinata	Max-Heap
100	3.06655e-07	4.08733e-06	1.73338e-07	3.044e-06
200	3.21339e-07	7.98001e-06	1.79325e-07	5.61400e-06
300	3.83332e-07	1.18193e-05	3.59328e-07	9.02000e-06
400	4.95336e-07	1.63280e-05	3.42671e-07	1.27840e-05
500	4.61334e-07	1.92893e-05	2.35996e-07	1.54447e-05
600	4.36000e-07	2.32333e-05	2.04658e-07	1.79347e-05
700	4.56008e-07	2.68753e-05	2.00666e-07	2.071e-05
800	4.81334e-07	3.04740e-05	2.23333e-07	2.41353e-05
900	4.89334e-07	3.43207e-05	2.37335e-07	2.60520e-05
1000	5.49336e-07	3.89587e-05	2.35996e-07	2.90487e-05
1100	5.07998e-07	4.1376e-05	2.32661e-07	3.17567e-05
1200	5.20679e-07	4.57427e-05	2.32668e-07	3.41660e-05
1300	5.71332e-07	4.93553e-05	2.46004e-07	3.73847e-05
1400	5.89327e-07	5.28720e-05	2.40002e-07	3.97167e-05
1500	6.01347e-07	5.68207e-05	2.48002e-07	4.38220e-05
1600	6.80004e-07	6.06173e-05	2.63345e-07	4.55907e-05
1700	8.62012e-07	6.66873e-05	2.71997e-07	4.97707e-05
1800	9.65336e-07	6.86440e-05	2.89331e-07	5.49e-05
1900	9.81997e-07	7.22307e-05	2.81987e-07	5.47033e-05
2000	7.89999e-07	7.54813e-05	2.77329e-07	5.72314e-05
2100	7.47341e-07	7.98320e-05	2.76663e-07	5.94100e-05
2200	7.60670e-07	8.42253e-05	2.63335e-07	6.24827e-05
2300	7.63334e-07	8.83500e-05	2.60662e-07	6.5436e-05
2400	7.09995e-07	9.07480e-05	2.76004e-07	6.70087e-05
2500	7.41322e-07	9.76060e-05	2.72668e-07	6.94660e-05
2600	7.28648e-07	1.00272e-04	2.80660e-07	7.31447e-05
2700	7.55996e-07	1.03333e-04	2.87336e-07	7.59133e-05
2800	8.08006e-07	1.06839e-04	2.69340e-07	7.88353e-05
2900	8.40005e-07	1.10371e-04	2.80002e-07	8.1134e-05
3000	8.56003e-07	1.15738e-04	2.65992e-07	8.327e-05
3100	8.21327e-07	1.18155e-04	2.68674e-07	8.78527e-05
3200	8.46003e-07	1.19927e-04	2.82e-07	9.12307e-05
3300	8.52662e-07	1.25855e-04	2.77328e-07	9.25080e-05
3400	8.41335e-07	1.29770e-04	2.65324e-07	9.54167e-05
3500	9.55987e-07	1.33235e-04	2.75333e-07	9.83647e-05
3600	9.16004e-07	1.35895e-04	2.90656e-07	1.00296e-04
3700	1.01601e-06	1.43341e-04	3.17992e-07	1.04555e-04
3800	9.65336e-07	1.44615e-04	2.99338e-07	1.05327e-04
3900	9.64670e-07	1.48681e-04	3.14661e-07	1.10985e-04
4000	9.97330e-07	1.53902e-04	3.08667e-07	1.12349e-04
4100	1.00266e-06	1.58757e-04	3.17339e-07	1.14166e-04
4200	9.8599e-07	1.59157e-04	3.08676e-07	1.16884e-04
4300	9.90001e-07	1.69956e-04	3.18672e-07	1.20035e-04
4400	9.97349e-07	1.72349e-04	3.18669e-07	1.22675e-04
4500	9.9067e-07	1.69558e-04	3.09995e-07	1.25839e-04
4600	9.66657e-07	1.70737e-04	3.06008e-07	1.27945e-04
4700	9.77994e-07	1.78925e-04	3.14657e-07	1.29699e-04
4800	9.68004e-07	1.80882e-04	3.42008e-07	1.33663e-04
4900	1.01600e-06	1.85257e-04	2.99998e-07	1.37385e-04
5000	1.02334e-06	1.88873e-04	3.16006e-07	1.37634e-04

Tabella 7: Tabella risultati della ricerca del minimo con valori decrescenti

8.3 OS-select

8.3.1 Valori random

	Min-heap	Lista ordinata	Lista non ordinata
100	1.43879e-04	2.60333e-06	1.71580e-04
200	3.37209e-04	4.69266e-06	5.65799e-04
300	6.25813e-04	7.38001e-06	1.29627e-03
400	9.71529e-04	1.02127e-05	2.59219e-03
500	1.19313e-03	1.1964e-05	3.41557e-03
600	1.56698e-03	1.49520e-05	5.12661e-03
700	1.82374e-03	1.6548e-05	6.35505e-03
800	2.11492e-03	1.81340e-05	8.05287e-03
900	2.57915e-03	2.19067e-05	1.05318e-02
1000	2.95121e-03	2.30987e-05	1.25299e-02
1100	3.19271e-03	2.45587e-05	1.47793e-02
1200	3.89517e-03	2.91327e-05	1.87149e-02
1300	3.85681e-03	2.75507e-05	1.96988e-02
1400	4.4573e-03	3.11973e-05	2.40501e-02
1500	4.97931e-03	3.40647e-05	2.83624e-02
1600	5.80758e-03	3.95967e-05	3.26970e-02
1700	6.48387e-03	4.19226e-05	3.69964e-02
1800	6.98003e-03	4.47513e-05	4.17257e-02
1900	7.14213e-03	4.38913e-05	4.40425e-02
2000	8.60158e-03	5.32727e-05	5.35386e-02
2100	8.07423e-03	4.75327e-05	5.22754e-02
2200	8.72629e-03	5.09020e-05	5.80087e-02
2300	9.72201e-03	5.58800e-05	6.84523e-02
2400	1.08388e-02	6.12793e-05	7.48238e-02
2500	1.04461e-02	5.64407e-05	7.32599e-02
2600	1.11940e-02	5.98367e-05	8.13170e-02
2700	1.14998e-02	5.98613e-05	8.41125e-02
2800	1.25742e-02	6.48287e-05	9.59951e-02
2900	1.49572e-02	7.6894e-05	1.10281e-01
3000	1.49041e-02	7.3024e-05	1.16609e-01
3100	1.58344e-02	7.80553e-05	1.22469e-01
3200	1.69416e-02	8.15573e-05	1.30560e-01
3300	1.66922e-02	7.83440e-05	1.29505e-01
3400	1.82817e-02	8.75547e-05	1.44655e-01
3500	1.63312e-02	7.52747e-05	1.37029e-01
3600	1.95743e-02	8.91307e-05	1.66539e-01
3700	2.18419e-02	9.81387e-05	1.77909e-01
3800	2.01750e-02	8.68493e-05	1.74316e-01
3900	2.28893e-02	9.91547e-05	1.90632e-01
4000	2.2479e-02	9.61347e-05	1.92827e-01
4100	2.39301e-02	9.914e-05	2.03571e-01
4200	2.55721e-02	1.04481e-04	2.16921e-01
4300	2.47688e-02	9.91860e-05	2.14875e-01
4400	2.8073e-02	1.16871e-04	2.41291e-01
4500	2.63807e-02	1.04141e-04	2.36758e-01
4600	2.92701e-02	1.16173e-04	2.70081e-01
4700	3.10044e-02	1.24435e-04	2.87433e-01
4800	3.29188e-02	1.30224e-04	3.00169e-01
4900	3.27705e-02	1.27017e-04	3.11054e-01
5000	3.39718e-02	1.27891e-04	3.15421e-01

Tabella 8: Tabella risultati di OS-select con valori random

8.3.2 Valori crescenti

	Min-heap	Lista ordinata	Lista non ordinata
100	1.36463e-04	2.45601e-06	1.63898e-04
200	3.52585e-04	4.97599e-06	6.13612e-04
300	5.88233e-04	6.90534e-06	1.27666e-03
400	9.43833e-04	1.07680e-05	2.56794e-03
500	1.25425e-03	1.25913e-05	3.66548e-03
600	1.47501e-03	1.41273e-05	4.86824e-03
700	1.83077e-03	1.67640e-05	6.56093e-03
800	2.23969e-03	1.97960e-05	8.71800e-03
900	2.47193e-03	2.09480e-05	1.03384e-02
1000	2.75768e-03	2.20507e-05	1.2385e-02
1100	3.39071e-03	2.63473e-05	1.612e-02
1200	3.66745e-03	2.80806e-05	1.83193e-02
1300	3.72453e-03	2.71347e-05	1.97965e-02
1400	4.72546e-03	3.42753e-05	2.55058e-02
1500	5.25451e-03	3.66553e-05	2.92563e-02
1600	5.46954e-03	3.69300e-05	3.19052e-02
1700	5.97903e-03	3.91180e-05	3.60633e-02
1800	6.81951e-03	4.45567e-05	4.16805e-02
1900	7.31575e-03	4.61667e-05	4.55375e-02
2000	7.90482e-03	4.82080e-05	5.1102e-02
2100	8.15125e-03	4.86033e-05	5.42028e-02
2200	8.62874e-03	5.1336e-05	5.88204e-02
2300	9.70634e-03	5.7946e-05	6.83948e-02
2400	9.67025e-03	5.48627e-05	6.89573e-02
2500	1.06482e-02	5.88833e-05	7.65003e-02
2600	1.14839e-02	6.3208e-05	8.50499e-02
2700	1.24385e-02	6.73873e-05	9.38004e-02
2800	1.26446e-02	6.62333e-05	9.89966e-02
2900	1.35453e-02	6.95200e-05	1.04097e-01
3000	1.40426e-02	7.03600e-05	1.14496e-01
3100	1.53673e-02	7.58293e-05	1.22967e-01
3200	1.56218e-02	7.48487e-05	1.23031e-01
3300	1.63051e-02	7.7842e-05	1.31809e-01
3400	1.88022e-02	8.797e-05	1.51814e-01
3500	1.9203e-02	8.90353e-05	1.61966e-01
3600	1.88988e-02	8.73387e-05	1.65371e-01
3700	1.92848e-02	8.686e-05	1.66612e-01
3800	2.16571e-02	9.17213e-05	1.84743e-01
3900	2.11517e-02	9.22406e-05	1.82672e-01
4000	2.14464e-02	8.98013e-05	1.86639e-01
4100	2.32459e-02	9.78547e-05	2.02207e-01
4200	2.55016e-02	1.05186e-04	2.21145e-01
4300	2.69756e-02	1.08522e-04	2.38751e-01
4400	2.7862e-02	1.12051e-04	2.48534e-01
4500	2.62420e-02	1.03091e-04	2.37631e-01
4600	2.83719e-02	1.12236e-04	2.69512e-01
4700	2.89865e-02	1.11509e-04	2.73812e-01
4800	3.09482e-02	1.20974e-04	2.89148e-01
4900	3.13634e-02	1.19606e-04	3.10418e-01
5000	3.27968e-02	1.20545e-04	3.11434e-01

Tabella 9: Tabella risultati di OS-select con valori crescenti

8.3.3 Valori decrescenti

	Min-heap	Lista ordinata	Lista non ordinata
100	1.52157e-04	2.57600e-06	1.75453e-04
200	3.71794e-04	4.87533e-06	6.13119e-04
300	6.20327e-04	7.00601e-06	1.40414e-03
400	9.89385e-04	9.67866e-06	2.50853e-03
500	1.22957e-03	1.19473e-05	3.96576e-03
600	1.66132e-03	1.50547e-05	5.45787e-03
700	2.0777e-03	1.8344e-05	7.30089e-03
800	2.29212e-03	1.94587e-05	8.87857e-03
900	2.78891e-03	2.2628e-05	1.14524e-02
1000	3.20765e-03	2.55693e-05	1.41504e-02
1100	3.25709e-03	2.46287e-05	1.52519e-02
1200	3.92767e-03	2.86834e-05	1.91203e-02
1300	4.35454e-03	3.21227e-05	2.23872e-02
1400	4.71217e-03	3.2776e-05	2.51805e-02
1500	5.21556e-03	3.47107e-05	2.90071e-02
1600	6.14778e-03	4.0578e-05	3.48365e-02
1700	6.59428e-03	4.34100e-05	3.84304e-02
1800	6.91173e-03	4.49513e-05	4.20032e-02
1900	7.21632e-03	4.4442e-05	4.5553e-02
2000	8.06531e-03	4.96973e-05	5.19642e-02
2100	8.48822e-03	4.95613e-05	5.61828e-02
2200	8.16462e-03	4.5758e-05	5.59680e-02
2300	9.87088e-03	5.56633e-05	6.93939e-02
2400	9.82710e-03	5.45167e-05	6.91802e-02
2500	1.07189e-02	5.80853e-05	7.60208e-02
2600	1.16613e-02	6.11660e-05	8.44498e-02
2700	1.22136e-02	6.21153e-05	9.02685e-02
2800	1.24726e-02	6.27533e-05	9.56519e-02
2900	1.32543e-02	6.67987e-05	1.00126e-01
3000	1.49758e-02	7.42080e-05	1.15376e-01
3100	1.53293e-02	7.41107e-05	1.1878e-01
3200	1.63870e-02	7.53953e-05	1.27848e-01
3300	1.73979e-02	8.19227e-05	1.35428e-01
3400	1.86426e-02	8.65753e-05	1.4878e-01
3500	1.72733e-02	7.6268e-05	1.43071e-01
3600	1.77569e-02	7.88753e-05	1.51363e-01
3700	2.01345e-02	8.91980e-05	1.67173e-01
3800	2.05168e-02	8.61840e-05	1.7511e-01
3900	2.13178e-02	9.40027e-05	1.81994e-01
4000	2.38450e-02	1.01422e-04	2.02081e-01
4100	2.36513e-02	9.88773e-05	2.03308e-01
4200	2.40233e-02	9.71827e-05	2.05451e-01
4300	2.64795e-02	1.05229e-04	2.27547e-01
4400	2.68536e-02	1.05663e-04	2.32812e-01
4500	2.80832e-02	1.09273e-04	2.48620e-01
4600	2.99424e-02	1.15937e-04	2.72122e-01
4700	3.10183e-02	1.23117e-04	2.87739e-01
4800	3.15316e-02	1.20933e-04	2.90077e-01
4900	3.25549e-02	1.22063e-04	3.04155e-01
5000	3.2942e-02	1.19485e-04	3.01981e-01

Tabella 10: Tabella risultati di OS-select con valori decrescenti

8.4 OS-rank

8.4.1 Valori random

	Min-heap	Lista ordinata	Lista non ordinata
100	1.19101e-04	2.79933e-06	1.64048e-04
200	2.87795e-04	5.46867e-06	5.88831e-04
300	4.91588e-04	7.68801e-06	1.25158e-03
400	7.75207e-04	1.1352e-05	2.36634e-03
500	9.54913e-04	1.28907e-05	3.50829e-03
600	1.14575e-03	1.52120e-05	4.72051e-03
700	1.46468e-03	1.756e-05	6.429e-03
800	1.74267e-03	2.01720e-05	8.29474e-03
900	2.09133e-03	2.3656e-05	1.09374e-02
1000	2.39592e-03	2.52987e-05	1.27758e-02
1100	2.58038e-03	2.86433e-05	1.50823e-02
1200	3.07262e-03	3.09227e-05	1.83336e-02
1300	3.34578e-03	3.26453e-05	2.10727e-02
1400	3.8014e-03	3.60247e-05	2.45594e-02
1500	4.22447e-03	4.03973e-05	2.90382e-02
1600	4.62553e-03	4.04800e-05	3.20030e-02
1700	5.04942e-03	4.59720e-05	3.66412e-02
1800	5.74671e-03	5.243e-05	4.32126e-02
1900	6.02974e-03	5.27173e-05	4.72786e-02
2000	6.65789e-03	5.54873e-05	5.23286e-02
2100	6.70567e-03	5.3076e-05	5.50445e-02
2200	7.41672e-03	5.7484e-05	6.12462e-02
2300	7.68538e-03	5.76047e-05	6.32926e-02
2400	8.49988e-03	6.61393e-05	7.23139e-02
2500	8.81026e-03	6.32646e-05	7.47579e-02
2600	9.44912e-03	6.69027e-05	8.19285e-02
2700	1.02437e-02	7.2108e-05	9.05146e-02
2800	1.1261e-02	8.16187e-05	1.02551e-01
2900	1.14668e-02	7.76980e-05	1.03675e-01
3000	1.20687e-02	8.03180e-05	1.14303e-01
3100	1.29074e-02	8.69867e-05	1.23902e-01
3200	1.38732e-02	8.61687e-05	1.34332e-01
3300	1.47278e-02	9.64327e-05	1.46083e-01
3400	1.46152e-02	9.41567e-05	1.41264e-01
3500	1.4472e-02	8.87127e-05	1.43016e-01
3600	1.62848e-02	9.94267e-05	1.62734e-01
3700	1.72448e-02	1.02169e-04	1.71947e-01
3800	1.80612e-02	1.02496e-04	1.80855e-01
3900	1.7835e-02	1.01329e-04	1.8835e-01
4000	1.92316e-02	1.09532e-04	1.98015e-01
4100	2.03539e-02	1.14263e-04	2.06486e-01
4200	2.12746e-02	1.18138e-04	2.17989e-01
4300	2.16774e-02	1.24565e-04	2.35749e-01
4400	2.33358e-02	1.24163e-04	2.4213e-01
4500	2.37983e-02	1.29625e-04	2.53935e-01
4600	2.28615e-02	1.21816e-04	2.51608e-01
4700	2.45148e-02	1.30638e-04	2.73429e-01
4800	2.46028e-02	1.26492e-04	2.68136e-01
4900	2.62829e-02	1.39759e-04	2.94003e-01
5000	2.91702e-02	1.41643e-04	3.18441e-01

Tabella 11: Tabella risultati di OS-rank con valori random

8.4.2 Valori crescenti

	Min-heap	Lista ordinata	Lista non ordinata
100	1.36463e-04	2.45601e-06	1.63898e-04
200	3.52585e-04	4.97599e-06	6.13612e-04
300	5.88233e-04	6.90534e-06	1.27666e-03
400	9.43833e-04	1.07680e-05	2.56794e-03
500	1.25425e-03	1.25913e-05	3.66548e-03
600	1.47501e-03	1.41273e-05	4.86824e-03
700	1.83077e-03	1.67640e-05	6.56093e-03
800	2.23969e-03	1.97960e-05	8.71800e-03
900	2.47193e-03	2.09480e-05	1.03384e-02
1000	2.75768e-03	2.20507e-05	1.2385e-02
1100	3.39071e-03	2.63473e-05	1.612e-02
1200	3.66745e-03	2.80806e-05	1.83193e-02
1300	3.72453e-03	2.71347e-05	1.97965e-02
1400	4.72546e-03	3.42753e-05	2.55058e-02
1500	5.25451e-03	3.66553e-05	2.92563e-02
1600	5.46954e-03	3.69300e-05	3.19052e-02
1700	5.97903e-03	3.91180e-05	3.60633e-02
1800	6.81951e-03	4.45567e-05	4.16805e-02
1900	7.31575e-03	4.61667e-05	4.55375e-02
2000	7.90482e-03	4.82080e-05	5.1102e-02
2100	8.15125e-03	4.86033e-05	5.42028e-02
2200	8.62874e-03	5.1336e-05	5.88204e-02
2300	9.70634e-03	5.7946e-05	6.83948e-02
2400	9.67025e-03	5.48627e-05	6.89573e-02
2500	1.06482e-02	5.88833e-05	7.65003e-02
2600	1.14839e-02	6.3208e-05	8.50499e-02
2700	1.24385e-02	6.73873e-05	9.38004e-02
2800	1.26446e-02	6.62333e-05	9.89966e-02
2900	1.35453e-02	6.95200e-05	1.04097e-01
3000	1.40426e-02	7.03600e-05	1.14496e-01
3100	1.53673e-02	7.58293e-05	1.22967e-01
3200	1.56218e-02	7.48487e-05	1.23031e-01
3300	1.63051e-02	7.7842e-05	1.31809e-01
3400	1.88022e-02	8.797e-05	1.51814e-01
3500	1.9203e-02	8.90353e-05	1.61966e-01
3600	1.88988e-02	8.73387e-05	1.65371e-01
3700	1.92848e-02	8.686e-05	1.66612e-01
3800	2.16571e-02	9.17213e-05	1.84743e-01
3900	2.11517e-02	9.22406e-05	1.82672e-01
4000	2.14464e-02	8.98013e-05	1.86639e-01
4100	2.32459e-02	9.78547e-05	2.02207e-01
4200	2.55016e-02	1.05186e-04	2.21145e-01
4300	2.69756e-02	1.08522e-04	2.38751e-01
4400	2.7862e-02	1.12051e-04	2.48534e-01
4500	2.62420e-02	1.03091e-04	2.37631e-01
4600	2.83719e-02	1.12236e-04	2.69512e-01
4700	2.89865e-02	1.11509e-04	2.73812e-01
4800	3.09482e-02	1.20974e-04	2.89148e-01
4900	3.13634e-02	1.19606e-04	3.10418e-01
5000	3.27968e-02	1.20545e-04	3.11434e-01

Tabella 12: Tabella risultati di OS-rank con valori crescenti

8.4.3 Valori decrescenti

	Min-heap	Lista ordinata	Lista non ordinata
100	1.25756e-04	3.34534e-06	1.79233e-04
200	3.02066e-04	5.624e-06	6.22012e-04
300	5.33305e-04	8.92400e-06	1.47028e-03
400	8.67932e-04	1.28367e-05	3.02213e-03
500	1.0244e-03	1.3874e-05	3.82548e-03
600	1.33314e-03	1.73400e-05	5.71834e-03
700	1.52599e-03	1.86107e-05	7.04404e-03
800	1.83241e-03	2.1722e-05	9.10678e-03
900	2.15188e-03	2.47047e-05	1.13690e-02
1000	2.29610e-03	2.42333e-05	1.27326e-02
1100	2.60705e-03	2.74093e-05	1.50472e-02
1200	3.08168e-03	3.05747e-05	1.88801e-02
1300	3.37547e-03	3.31600e-05	2.12626e-02
1400	3.49338e-03	3.23913e-05	2.2795e-02
1500	4.13341e-03	3.92693e-05	2.87470e-02
1600	4.57299e-03	4.05820e-05	3.21451e-02
1700	5.19433e-03	4.43020e-05	3.7151e-02
1800	5.67359e-03	4.65767e-05	4.19885e-02
1900	6.11115e-03	4.91320e-05	4.61352e-02
2000	6.49993e-03	5.10033e-05	5.01205e-02
2100	7.16069e-03	5.82127e-05	6.02283e-02
2200	7.36600e-03	5.7086e-05	6.12337e-02
2300	8.07551e-03	5.87593e-05	6.63278e-02
2400	8.90490e-03	6.5848e-05	7.55692e-02
2500	9.16819e-03	6.69413e-05	7.97098e-02
2600	1.0264e-02	7.21333e-05	8.96579e-02
2700	1.03434e-02	6.86767e-05	9.01154e-02
2800	1.12057e-02	7.27413e-05	9.93008e-02
2900	1.15683e-02	7.11780e-05	1.01320e-01
3000	1.19993e-02	7.49833e-05	1.09728e-01
3100	1.29972e-02	8.30693e-05	1.24164e-01
3200	1.45068e-02	8.60480e-05	1.37159e-01
3300	1.52992e-02	8.8078e-05	1.45245e-01
3400	1.54388e-02	9.30827e-05	1.50290e-01
3500	1.40500e-02	8.10107e-05	1.36351e-01
3600	1.60911e-02	9.48660e-05	1.59053e-01
3700	1.68044e-02	9.62247e-05	1.65492e-01
3800	1.80286e-02	9.9118e-05	1.84405e-01
3900	1.89468e-02	1.07244e-04	1.93437e-01
4000	1.92656e-02	1.04787e-04	1.93147e-01
4100	2.10654e-02	1.17615e-04	2.29366e-01
4200	2.13868e-02	1.10993e-04	2.14633e-01
4300	2.31095e-02	1.23415e-04	2.45763e-01
4400	2.45917e-02	1.25264e-04	2.47433e-01
4500	2.29252e-02	1.14744e-04	2.36071e-01
4600	2.34254e-02	1.12976e-04	2.52303e-01
4700	2.52045e-02	1.22429e-04	2.73880e-01
4800	2.67683e-02	1.30044e-04	2.88193e-01
4900	2.79424e-02	1.30022e-04	2.95756e-01
5000	2.99988e-02	1.44833e-04	3.22348e-01

Tabella 13: Tabella risultati di OS-rank con valori decrescenti