Relazione strutture dati Heap, LikedList, LinkedOrderedList

1. Obiettivo
2. Premesse teoriche
3. Tempi teorici
4. Schema dei moduli e delle classi
5. Scelte implementate
6. Dati utilizzati
7. Specifiche della piattaforma di test
8. Misurazioni effettuate
9. Presentazione dei risultati
10. Conclusione, prima della documentazione
11. Documentazione
12. heap.py
13. linkedList.py
14. linkedOrderedList.py
15. test.py

Obiettivo

Facendo riferimento ai principi studiati nel corso di Algoritmi e Strutture Dati, l'obiettivo è valutare sperimentalmente le prestazioni delle strutture dati Heap, LinkedList e LinkedOrderedList, scegliendo test che riflettano le loro caratteristiche teoriche più rilevanti (ad esempio, tempi di inserimento, cambio valore (quindi anche ricerca) e estrazione max/min).

Premesse Teoriche

Heap:

- Struttura dati ad albero binario completo, quindi può essere rappresentato tramite array
- Al nodo i-esimo abbiamo:
  - Parent(i) = $\lfloor i/2 \rfloor$
  - Left(i) = $2i$
  - Right(i) = $2i + 1$
- Se maxHeap => A[parent] $\geq$ A[child]
- Se minHeap => A[parent] $\leq$ A[child]
- Il primo elemento è massimo se maxHeap (minimo se minHeap)
- L'altezza massima dell'albero binario heap è $h = \lfloor \log_2 n \rfloor$

Linked List:

- Composta da nodi (single linked list) dove si ha un unico puntatore al nodo successivo
- Non ha alcun ordine, per cui generalmente ci si aspettano dei tempi e costi maggiori di tutte le altre strutture dati in questo esercizio
- L'inserimento è l'unico vantaggio se fatto in testa (non è il nostro caso per scelta implementativa così da preservare gli indici dei valori), non avendo alcun ordine o criterio le operazioni di inserimento sono le più immediate

- Per la ricerca ci si aspetta di dover scorrere l'intera lista nel caso peggiore quindi costo lineare, vedi tabella tempi teorici sotto

Linked Ordered List:

- Composta da nodi come la Linked List con unico puntatore al nodo successivo
- Gli elementi sono in ordine decrescente se maxLinkedOrderedList così da avere massimo come nodo `root` oppure minimo se altrimenti, questo riduce notevolmente i costi di estrazione del massimo/minimo
- L'inserimento deve scorrere la lista finchè non trova il predecessore per concatenare il nuovo nodo, a differenza della lista concatenata non ordinata che nel nostro caso deve concatenare in fondo alla lista

Tempi teorici

Struttura	Insert	Inc/Dec	Extract
Heap	$O(\log n)$ (costo bubble-up)	$O(\log n)$ (costo bubble-up)	$O(\log n)$ (costo heapify dell'ultimo valore messo in testa)
Lista concatenata	$O(n)$ (inserimento in coda)	$O(n)$ (ricerca elemento per indice)	$O(n)$ (ricerca valore massimo)
Lista concatenata ordinata	$O(n)$ (inserimento in ordine)	$O(n)$ (ricerca elemento per indice)	$O(1)$ (è il primo elemento)

Schema dei moduli e delle classi

L'unico modulo o package che è stato creato e quello per la cartella data_structures per una facile importazione nel file test.py. Gli elementi di suddetto package sono le seguenti classi (vi sono altri attributi come il nome e il tipo ma servono solo per un migliore riconoscimento su test.py):

- **Heap** : Attributi:
  - `array` : int = lista python che contiene i valori.
  - `type` : string = può essere "max" oppure "min", usato per non ripetere il codice.
  - `size` = inizialmente zero. Funzioni:
  - `_heapify(self, i: int)` = serve a riordinare l'array con i padri e figli corretti dall'alto verso il basso.
  - `extract(self) -> int` = estrae il massimo o il minimo, che è sempre il primo elemento per metterci nella posizione l'ultimo elemento ed infine chiama heapify() per ripristinarne l'ordine.
  - `checkOrder(seld, i=0) -> bool` = controlla dopo ogni operazione di inserimento (quindi anche bubble-up), incremento o decremento di un valore e di estrazione se l'ordine è stato preservato o ristabilito.
  - `incDecValue(self, i: int, value: int) -> bool` = Cambia il valore dell'elemento all'indice i e riordina l'heap se necessario (bubble-up o bubble-down a seconda del tipo di heap).
- **LinkedList** : Attributi:

    - `head` : riferimento al primo nodo della lista.
    - `size` : numero di elementi nella lista. Funzioni:
    - `insert(self, value: int)` : inserisce un nuovo nodo in coda alla lista.
    - `incDecValue(self, i: int, value: int) -> bool` : modifica il valore dell'elemento all'indice i (scorrendo la lista) senza riordino.
    - `extract(self) -> int` : cerca e rimuove il valore massimo (scorrendo tutta la lista).
- **LinkedOrderedList** : Attributi:
    - `head` : riferimento al primo nodo della lista ordinata.
    - `size` : numero di elementi nella lista. Funzioni:
    - `insert(self, value: int)` : inserisce il nuovo valore mantenendo l'ordine crescente/decrescente.
    - `incDecValue(self, i: int, value: int) -> bool` : modifica il valore dell'elemento all'indice i (scorrendo la lista) e riordina la lista se necessario.
    - `checkOrder(seld, i=0) -> bool` = controlla dopo ogni operazione di inserimento, incremento o decremento di un valore e di estrazione se l'ordine è stato preservato o ristabilito, in questo caso, se l'ultimo elemento è il pi.
    - `extract(self) -> int` : estrae e rimuove il primo elemento (che è sempre il massimo o minimo, a seconda dell'ordinamento).

Scelte implementative

- Heap è stato implementato con una **lista python**, poiché simula il comportamento come un array così da essere più efficiente.
- La funzione di **inserimento** nelle lista non ordinata inserisce gli elementi in **coda** per evitare di slittare gli indici degli elementi.
- La funzione di **cambio valore** prevede sia uno **scorrimento** per indice che un **riordinamento** (*tranne nel caso della lista collegata non ordinata*).
- La funzione di estrazione prevede la ricerca del valore massimo nel caso della lista collegata non ordinata.
- Inc/Dec si riferisce all'operazione di cambio valore dell'elemento nella struttura dati che però nel caso dell'heap deve poter solo incrementare il valore se maxHeap altrimenti decrementare se minHeap (per costruzione dell'algoritmo di bubble-up da libro).

Dati utilizzati

- Dimensioni dei dataset
- Valore massimo per il range della generazione randomica
- Valore massimo di incremento/decremento
- I valori dei dataset vengono generati automaticamente tramite funzioni randomiche del package rand, per garantire riproducibilità e variabilità ogni volta che si esegue il programma test.py.
  Ad ogni iterazione, seppur la dimensione rimane la stessa, gli elementi dell'array sono di nuovo randomici rispetto al dataset della iterazione precedente.
- Ogni operazione, iterazione e dimensione ha il relativo dataset di numeri randomici, così da avere tempi e misurazioni indipendenti facendone la media, nel caso attuale 50 iterazioni per operazione vengono fatte per avere una media dei tempi sufficientemente accurata con le sperimentazioni.
- In alternativa, si possono utilizzare dataset reali per valutare le prestazioni su casi concreti. **NB:** alcune misurazioni potrebbero essere influenzate dal thrashing della memoria, per ovviare a ciò quindi il numero di iterazioni è stato scelto sufficientemente alto `numberOfIterations = 50` da

ridurre tali "rumori" che possono sporcare le osservazioni sperimentali dell'andamento della curva dei tempi nei grafici.

Specifiche della piattaforma di test

- **Hardware**:
  - CPU = Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz, 1800 Mhz, 4 Core(s), 8 Logical Processor(s)
  - RAM = Installed Physical Memory (RAM) 8,00 GB
  - Modello = LENOVO_MT_81HN_BU_idea_FM_V130-15IKB
- **Sistema operativo**: Windows 11 Home

Misurazioni effettuate

- Tutte le misurazioni sono state effettuate sulla stessa macchina per garantire coerenza.
- Sono stati misurati i tempi di esecuzione singoli delle principali operazioni (inserimento, cambio valore, estrazione) su insiemi di dati di dimensione arbitraria ma sufficientemente grande, di seguito sono illustrati grafici con dimensioni del dataset dei tempi medi calcolati su `numberOfIterations = 50`.
- I tempi singoli e medi sono stati salvati in dataframe esportati poi su file .csv sotto `/tables/`.
- La mappa meanTimingLists è un calcolo medio, della somma dei tempi di timingLists a partire dalla prima iterazione fino alla i-esima della operazione specifica sul dataset specifico. Ho scelto di usare tale metrica per rimuovere rumori e punti di discontinuità nei grafici dei tempi così che sia ancora più evidente l'**andamento** all'aumentare della dimensione del dataset piuttosto che i tempi aggregati singoli.
- Ogni test quindi genera valori randomici secondo i parametri di input o di default.

Presentazione dei risultati

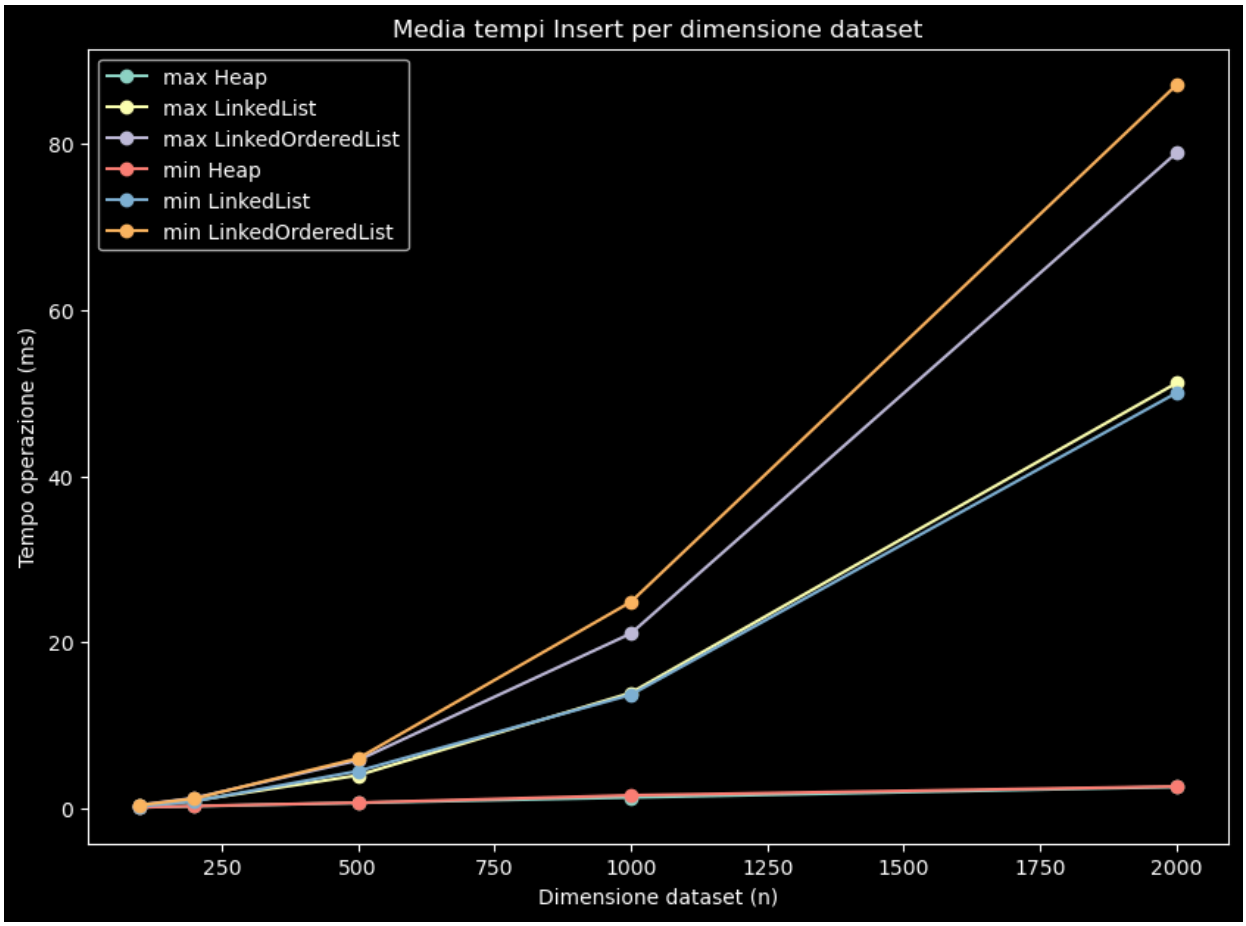
Inizialmente avevo sfruttato la funzione `time()` del package `time`, notando però notevoli picchi ogniqualvolta la memoria del mio sistema andava in thrashing; il garbage collector subentrava in quelle frazioni di esecuzione dei vari metodi delle strutture dati così da aggregare anche il tempo di esecuzione della routine del garbage collector -> noi **non** vogliamo questo per una misurazione e comparazione pulita dei tempi.

Per ovviare a tale problema ho sfruttato la funzione `gc.disable()` di `gc` (garbage collector) che evita di far sì che subentri la prelazione del garbage collector durante l'esecuzione delle operazioni.

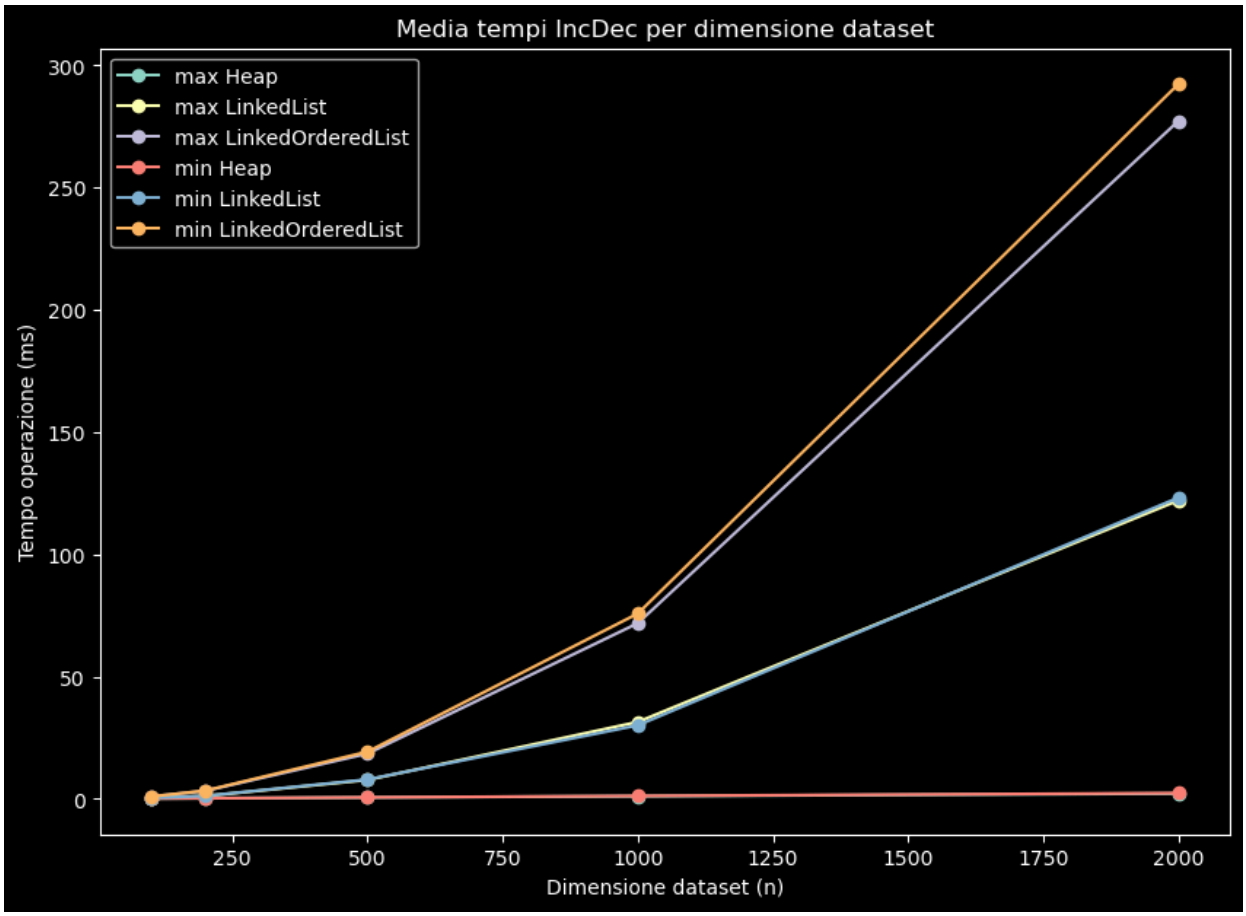
Osservazioni sperimentali

Possiamo osservare:

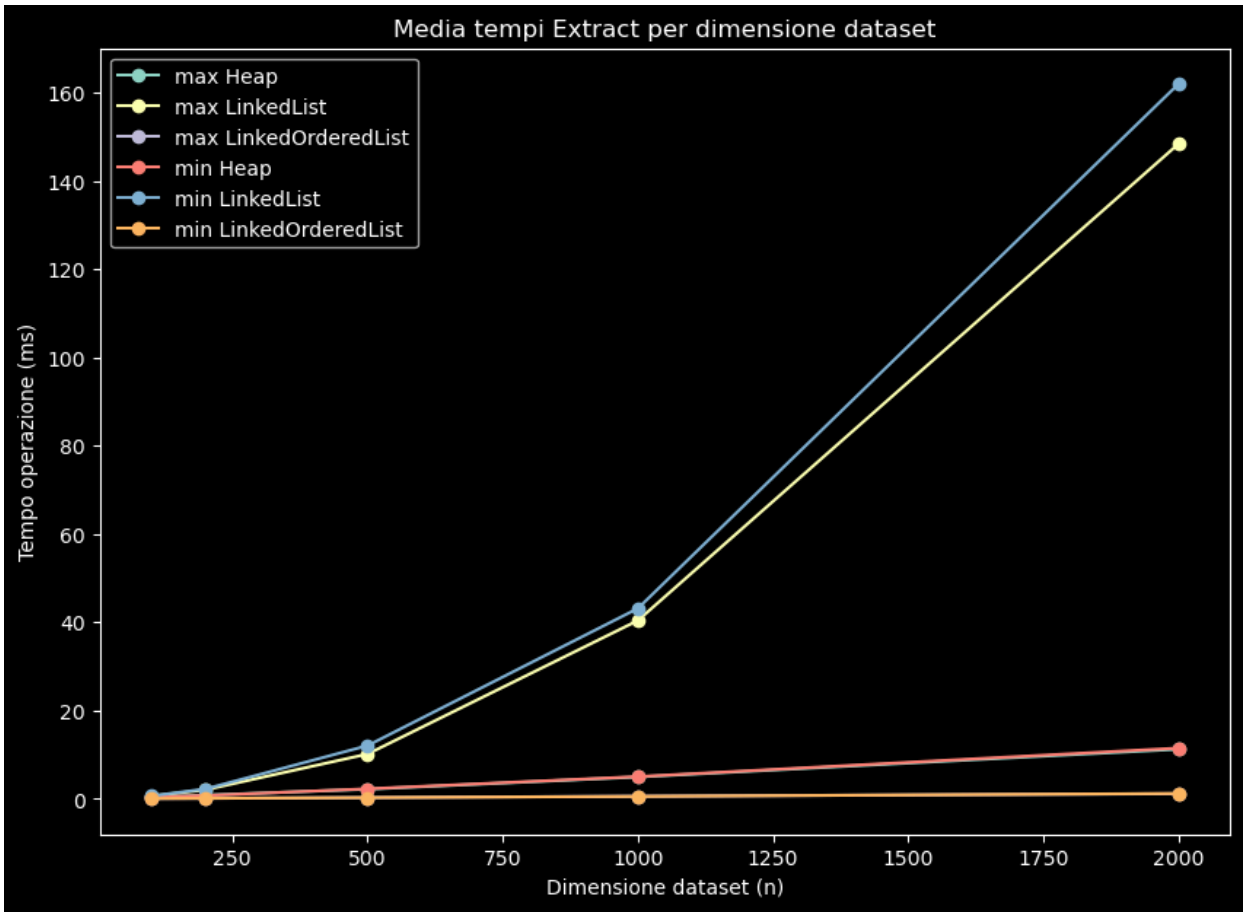
1. **Insert**



2. **IncDec**



3. **Extract**



Conclusione

I test confermano la teoria: Heap mantiene tempi logaritmici e costanti anche su grandi dataset, LinkedList scala linearmente ed è inefficiente per operazioni su molti dati, LinkedOrderedList è lenta all'inserimento ma velocissima in estrazione, LinkedList per l'inserimento è esponenziale perché viene fatto in code -> si deve scorrere per ottenere l'ultimo puntatore a nodo non nullo. La scelta della struttura deve seguire la frequenza e il tipo di operazioni richieste: Heap primeggia nella gestione di priorità e grandi numeri, le liste vanno bene solo per casi piccoli o accessi sequenziali.

Documentazione

Di seguito sono riportati i frammenti di codice presi dalla cartella `data_structures` e da `test.py`, con una breve spiegazione immediatamente sotto ogni blocco.

File: `data_structures/heap.py`

```python
class Heap:
    def __init__(self,type: str):
        self.array = []
        self.type = type
        self.size = 0
```

Spiegazione: costruttore della classe Heap; inizializza l'array che rappresenta l'heap, il tipo ("max" o "min") e la dimensione `size`.

```python
def _heapify(self, i: int):
    l = i * 2 + 1 # figlio sinistro considerando i che parte da zero
    r = i * 2 + 2 # figlio destro
    basket = i # cestello che contiene indice max/min da scambiare
```

```
    if l < len(self.array) and (self.array[l] > self.array[i] if self.type == "max"
else self.array[l] < self.array[i]):
        basket = l # massimo attuale
    if r < len(self.array) and (self.array[r] > self.array[basket] if self.type ==
"max" else self.array[r] < self.array[basket]):
        basket = r
    if basket != i:
        tmp = self.array[i]
        self.array[i] = self.array[basket]
        self.array[basket] = tmp
        self._heapify(basket)
```

Spiegazione: funzione ricorsiva che ripristina la proprietà di heap (heapify / bubble-down) a partire dall'indice `i`, scambiando con il figlio massimo/minimo a seconda di `self.type`.

```
def extract(self) -> int: # estrae il massimo/minimo
    if len(self.array) < 1:
        return None
    basket = self.array[0] # basket contiene o il max o il min
    self.array[0] = self.array[-1] # mette come primo elemento (max/min) il valore
finale
    self.array.pop() # elimina l'elemento all'ultima posizione che sarebbe
l'elemento massimo dopo lo scabio della riga sopra
    self._heapify(0)
    self.size -= 1
    return basket
```

Spiegazione: estrae il valore della radice (massimo o minimo), sostituisce la radice con l'ultimo elemento, rimuove l'ultimo e richiama `_heapify(0)` a partire dalla radice per ripristinare la proprietà dell'heap.

```
def incDecValue(self, i: int, value: int) -> bool: # incrementa/diminuisce il valore
a seconda del tipo di heap
    if (value < self.array[i] if self.type == "max" else value > self.array[i]):
        return False # se il tipo è max/min il valore deve per forza essere
maggiore/minore o uguale rispettivamente
    self.array[i] = value
    parent = (i - 1) // 2
    while i > 0 and (self.array[parent] < self.array[i] if self.type == "max" else
self.array[parent] > self.array[i]):
        tmp = self.array[parent]
        self.array[parent] = self.array[i]
        self.array[i] = tmp
        i = parent
        parent = (i - 1) // 2
    return True
```

Spiegazione: modifica il valore alla posizione `i` e, se necessario, effettua il bubble-up (scambia con i padri) per ripristinare la proprietà di heap; verifica anche che la nuova valore sia coerente con il tipo di heap.

```
def insert(self, value: int):
    self.array.append(value)
    self.incDecValue(len(self.array) - 1, value)
    self.size += 1
```

Spiegazione: inserisce `value` aggiungendolo in coda all'array e chiama `incDecValue` per posizionarlo correttamente (bubble-up); aggiorna `size`.

File: `data_structures/linkedList.py`

```
class LinkedList:
    def __init__(self, type):
        self.root = None
        self.size = 0
        self.type = type
```

Spiegazione: costruttore della lista concatenata non ordinata; `root` punta al primo nodo, `size` mantiene il conteggio, `type` indica se si considera massimo o minimo nelle operazioni (usato in `extract`).

```
def insert(self, value):
    node = Node(value)
    if self.size == 0 or self.root is None:
        self.root = node
    else:
        lastNode = self._getLast()
        lastNode.next = node
    self.size += 1
```

Spiegazione: inserisce un nuovo nodo in coda alla lista; se la lista è vuota imposta il nuovo nodo come `root`, altrimenti appende alla fine.

```
def _getLast(self):
    if self.size == 0:
        return None
    lastNode = self.root
    while lastNode.next is not None:
        lastNode = lastNode.next
    return lastNode
```

Spiegazione: ritorna l'ultimo nodo della lista scorrendo i puntatori `next`.

```
def _getNode(self, i):
    if self.size == 0:
        return None
    targetNode = self.root
    while targetNode is not None and i > 0:
        targetNode = targetNode.next
        i -= 1
    return targetNode
```

Spiegazione: ritorna il nodo alla posizione indicata (indice zero-based) scorrendo dalla radice.

```
def extract(self): # estrae il massimo o il minimo
    if self.size == 0 or self.root is None:
        return None
    predecessorTargetNode = None
    targetNode = self.root # nodo di appoggio max o min
    predecessorNode = None
    node = self.root
    while node is not None:
        if (node.value > targetNode.value if self.type == "max" else node.value <
targetNode.value):
            predecessorTargetNode = predecessorNode
            targetNode = node
        predecessorNode = node
        node = node.next
    if predecessorTargetNode is not None:
        predecessorTargetNode.next = targetNode.next
    else:
        self.root = targetNode.next
    self.size -= 1
    return targetNode
```

Spiegazione: scorre tutta la lista per trovare il nodo con valore massimo (o minimo) a seconda di `self.type`, rimuove quel nodo collegando il predecessore al successivo e lo restituisce.

```
def incDecValue(self, i, value): # non necessita alcun controllo sui valori (maggiore dell'attuale se max, minore se min) perché è disordinata la lista
    if i < 0 or i >= self.size or self.size == 0:
        return False
    targetNode = self._getNode(i)
    if targetNode is None:
        return False
    targetNode.value = value
    return True
```

Spiegazione: cambia il valore del nodo alla posizione `i` senza ulteriori riordini (lista non ordinata), verificando i limiti.

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

Spiegazione: classe nodo semplice con `value` e puntatore `next`.

File: `data_structures/linkedOrderedList.py`

```
class LinkedOrderedList:
    def __init__(self, type):
        self.root = None
        self.size = 0
        self.type = type
```

Spiegazione: costruttore della lista collegata ordinata; `type` indica se l'ordine è crescente/decrescente.

```
def insert(self, value):
    node = Node(value)
    if self.size == 0:
        self.root = node
    elif (node.value > self.root.value if self.type == "max" else node.value < self.root.value):
        node.next = self.root
        self.root = node
    else:
        predecessorNode = self._getPredecessor(node.value)
        node.next = predecessorNode.next
        predecessorNode.next = node
    self.size += 1
```

Spiegazione: inserisce `value` mantenendo l'ordine richiesto; se deve essere primo lo posiziona come `root`, altrimenti trova il predecessore e lo inserisce dopo di lui.

```
def _getPredecessor(self, value): # serve a cercare l'ultimo nodo più grande/piccolo di quello da inserire
    predecessorNode = self.root
    while predecessorNode.next is not None and (value < predecessorNode.next.value if self.type == "max" else value > predecessorNode.next.value):
        predecessorNode = predecessorNode.next
    return predecessorNode
```

Spiegazione: cerca il nodo dopo il quale inserire il nuovo valore.

```
def extract(self): # estrae il massimo o il minimo che sarà sempre il primo elemento poiché la lista è ordinata
```

```
    if self.size == 0:
        return None
    targetNode = self.root
    if self.root.next is not None:
        self.root = self.root.next
    else:
        self.root = None
    self.size -= 1
    return targetNode # estraendo il primo elemento la lista rimarrà sempre ordinata
```

Spiegazione: rimuove e restituisce il primo nodo (`root`) che è il massimo/minimo a seconda dell'ordinamento: operazione O(1).

```
def incDecValue(self, i, value): # quando cambio il valore a un singolo nodo basta che riordini quest'ultimo e non l'intera lista
    if i < 0 or i >= self.size or self.size == 0:
        return False
    predecessorNode = None # puntatore predecessore per lo scorrimento
    node = self.root # puntatore nodo scorrimento
    while node is not None and i > 0:
        predecessorNode = node
        node = node.next
        i -= 1
    if predecessorNode is not None:
        predecessorNode.next = node.next  # tolgo il nodo alla posizione corrente
    else:
        self.root = node.next # se il predecessore non c'è allora il nodo diventa il nuovo root
    node.value = value
    if self.root is None or (node.value > self.root.value if self.type == "max" else node.value < self.root.value):
        node.next = self.root
        self.root = node
    else:
        predecessorNewNode = self._getPredecessor(value)
        node.next = predecessorNewNode.next
        predecessorNewNode.next = node
    return True
```

Spiegazione: cambia il valore del nodo all'indice `i` , lo rimuove dalla posizione corrente e lo reinserisce nella posizione corretta per mantenere l'ordine; restituisce True se l'operazione ha successo.

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

Spiegazione: definizione del nodo per la lista ordinata.

File: `test.py` (parti principali)

```
import random, time
from tabulate import tabulate
import matplotlib.pyplot as plt
from data_structures import *
import pandas as pd
```

Spiegazione: import delle librerie usate nello script di test: `random` e `time` per generazione e misurazioni, `tabulate` e `matplotlib` per output/grafici, importa le strutture da `data_structures` e `pandas` per salvare CSV.

```
### RANDOM NUMBER SET ###
while True:
    numberSetSize = int(input("Inserire il numero di dati randomici da testare
[default = 500]-> ") or 500)
    if numberSetSize > 0:
        break
while True:
    maxValue = int(input("Inserire il valore massimo dei numeri [default = 500]-> ")
or 500)
    if maxValue > 0:
        break
numbers = random.sample(range(0, maxValue), numberSetSize)
while True:
    numberIncDecLimit = int(input("Inserire il valore massimo di
incremento/decremento [default = 500]-> ") or 500)
    if numberIncDecLimit > 0:
        break
#########################
```

Spiegazione: acquisisce i parametri del test dall'utente (numero di elementi, range dei valori, limite per inc/dec) e genera la lista `numbers` di campioni casuali.

```
### DATA STRUCTURES ###
structures = {
    'maxHeap': Heap("max"),
    'minHeap': Heap("min"),
    'maxLinkedList': LinkedList("max"),
    'minLinkedList': LinkedList("min"),
    'maxLinkedOrderedList': LinkedOrderedList("max"),
    'minLinkedOrderedList': LinkedOrderedList("min"),
}
#######################
```

Spiegazione: crea le istanze delle strutture che verranno testate (heap e liste, per max e min).

```
### OPERATIONS ###
operations = ['Insert', 'IncDec', 'Extract']
##################
```

Spiegazione: definisce la lista delle operazioni da cronometrare.

```
### TIMING LISTS ###
timingLists = {
    f"{name}{op}Times": [] for name in structures for op in operations
}

sumTimingLists = {
    f"{name}{op}Times": [] for name in structures for op in operations
}
####################
```

Spiegazione: inizializza i dizionari per memorizzare i tempi singoli e cumulativi per ogni struttura e operazione.

```
### insert ###
print("Test insert:\n")
i = 0
for n in numbers:
    for name, ds in structures.items():
        i += 1
        start = time.time()
        flag = ds.insert(n)
        end = time.time()
```

```
        elapsed = (end - start) * 1000 # s * 1000 = ms
        timingLists[f"{name}InsertTimes"].append(elapsed)
        if flag == False:
            raise(Exception(f""))
        print(f"{name} -> inserito valore = {n} iterazione = {i} in {elapsed}ms")
##############
```

Spiegazione: ciclo di inserimento che inserisce ogni valore di `numbers` in tutte le strutture, misura il tempo e salva i tempi.

```
### incDecValue ###
print("Test inc/dec valore:\n")
i = 0
for n in numbers:
    for name, ds in structures.items():
        i += 1
        randomIndex = random.randrange(0, numberSetSize)
        randomValue = random.randrange(0, numberIncDecLimit)
        if name[3:7] == "Heap":
            value = ds.getValue(randomIndex)
            if name[0:3] == "max":
                value += randomValue
            else:
                value -= randomValue
        else:
            value = randomValue
        start = time.time()
        flag = ds.incDecValue(randomIndex, value)
        end = time.time()
        if flag == False:
            raise Exception(f"{name} -> errore (iterazione {i} nell'indice
{randomIndex} fuori range, dimensione[{ds.size}] o struttura vuota!")
        elapsed = (end - start) * 1000 # ms * 1000 = ms
        timingLists[f"{name}IncDecTimes"].append(elapsed)
        print(f"{name} -> cambiato valore = {value} iterazione = {i} in
{elapsed}ms")
####################
```

Spiegazione: per ogni struttura sceglie un indice e un valore casuale; per gli heap modifica il valore coerentemente (aumenta per max, diminuisce per min), esegue `incDecValue` e registra il tempo.

```
### extract ###
print("Test extract valori max/min:\n")
i = 0
for n in numbers:
    for name,ds in structures.items():
        i += 1
        start = time.time()
        node = ds.extract()
        end = time.time()
        elapsed = (end - start) * 1000 # ms * 1000 = ms
        timingLists[f"{name}ExtractTimes"].append(elapsed)
        if node is None:
            raise Exception(f"{name} -> errore durante l'estrazione {i}, la
struttura è vuota!")
        else:
            print(f"{name} -> estratto nodo {(node if isinstance(node, int) else
node.value)} iterazione = {i} in {elapsed}ms")
###############
```

Spiegazione: esegue le estrazioni ripetute su ogni struttura, misura i tempi e verifica che l'estrazione sia avvenuta correttamente.

```
### CALCULATING THE TOTAL TIME PER ITERATION ###
for op in operations:
    for name, ds in structures.items():
        prev = 0
        for t in timingLists[f"{name}{op}Times"]:
            current = t # copia per valore dato che int è una variabile immutabile
            current += prev
            prev = current
            sumTimingLists[f"{name}{op}Times"].append(current)
################################################
```

Spiegazione: costruisce i tempi cumulativi (somma progressiva) a partire dai tempi singoli per ogni struttura e operazione.

```
### SAVING CSV FILE ###
# IN THE "timingLists_combaned.csv" file, format is the following #
# LEFT TABLE -> SINGLE TIME TABLE #
# RIGHT TABLE -> SUM TIME TABLE #
timing_df = pd.DataFrame(timingLists)
sum_timing_df = pd.DataFrame(sumTimingLists)

max_len = max(len(timing_df), len(sum_timing_df))
timing_df = timing_df.reindex(range(max_len))
sum_timing_df = sum_timing_df.reindex(range(max_len))

timing_df.insert(0, "Iteration", range(1, max_len + 1))
sum_timing_df.insert(0, "Iteration", range(1, max_len + 1))

blank_col = pd.Series([""] * max_len, name="")

combined_df = pd.concat(
    [timing_df, blank_col, sum_timing_df.drop(columns="Iteration")],
    axis=1
)

combined_df.to_csv("tables/timingLists_combined.csv", index=False)
#########################
```

Spiegazione: converte i dizionari in DataFrame, li allinea e salva un file CSV combinato con tempi singoli e cumulativi.

```
### GRAPHS AND PLOTTING ###
plt.style.use("dark_background")
for op in operations:
    plt.figure(figsize=(10, 6))
    for name in structures:
        times = sumTimingLists[f"{name}{op}Times"]
        plt.plot(times, label=f"{name} {op}")
    plt.xlabel("Iterazioni")
    plt.ylabel("Tempo (ms)")
    plt.title(f"Tempi totali di {op} dati per ogni struttura")
    plt.legend()
    plt.savefig(f"graphs/{op}TotalTimes.png")
    plt.get_current_fig_manager().set_window_title(f"Grafico Tempi Totali")

for op in operations:
    plt.figure(figsize=(10, 6))
    for name in structures:
        times = timingLists[f"{name}{op}Times"]
        plt.bar(range(len(times)), times, label=f"{name} {op}", alpha=0.7, width=1)
    plt.xlabel("Iterazioni")
```

```
    plt.ylabel("Tempo (ms)")
    plt.title(f"Tempi singoli di {op} dati per ogni struttura")
    plt.legend()
    plt.savefig(f"graphs/{op}SingleTimes.png")
    plt.get_current_fig_manager().set_window_title(f"Grafico Tempi Singoli")

plt.show()
############################
```

Spiegazione: genera e salva i grafici (linee per tempi cumulativi e barre per tempi singoli) e mostra le figure.