Progetto #1

Abbiamo implementato la classe MyList presente all'interno del modulo my_list, usando come rappresentazione interna una lista doppiamente concatenata, e dando ai metodi di questa classe l'esatto comportamento della classe list ufficiale di Python.

La classe MyList eredita da una classe astratta che abbiamo chiamato DoubleLinkedList e che implementa l'ADT List.

DoubleLinkedList contiene tutti i metodi richiesti dalla traccia del progetto, che abbiamo poi implementato in MyList, non contiene ovviamente gli operatori e funzioni built-in richieste dalla traccia in quanto queste sono specifiche della classe MyList.

All'interno di MyList vi è una classe privata _Node, questa rappresenta il singolo nodo, e contiene come attributi left, right e element. Left e right consentono di muoversi a destra e a sinistra nella lista doppiamente linkata, element contiene l'elemento memorizzato in una certa posizione della lista.

In questa classe _Node si è usata la proprietà slots per fare in modo che Python non usi la sua rappresentazione a dizionario per pochi attributi la quale è eccessiva.

La nostra lista doppiamente linkata è costituita da due nodi iniziali e finali detti head e tail, i quali indicano rispettivamente il primo e l'ultimo elemento, questi sono inizializzati nel costruttore.

Nel costruttore è anche inizializzato _size (dimensione della lista) e _reverse (indica se la lista è stata invertita).

Al costruttore della class MyList proprio come la classe ufficiale se gli viene passato un oggetto iterabile genera la lista a partire da questo, se invece viene passato un oggetto non valido solleva un'eccezione.

Gestire il reverse, richiederebbe una complessità almeno di $O(n)$, in quanto sul posto andrebbero invertiti tutti i nodi, magari partendo dal centro. Questo è il motivo per cui nella classe abbiamo previsto l'attributo _reverse, questo indica se la lista è stata invertita oppure no, e ci consentirà

di sfruttare le proprietà di una lista doppiamente linkata, cambiando quindi la direzione di visita della lista e permettendo di ottenere col reverse complessità computazione di $\theta(1)$, in quanto basta scambiare testa e coda e settare la variabile flag _reverse.

Fare il reverse in questo modo ha ovviamente come svantaggio che in ogni altra funzione, che richiede la visita della lista è necessario verificare prima se è stata invertita. Il vantaggio però è ovvio, visto che la nostra struttura dati è pensata per operare con ampie quantità di dati, fare un controllo su se la lista è stata invertita può richiedere al massimo $\theta(1)$, mentre invertire una lista sul posto richiederebbe $O(n)$ e questo farebbe perdere tutto il senso di usare una lista doppiamente linkata.

Il metodo append, crea un nuovo nodo con l'elemento passato e lo aggiunge alla fine della lista.

Extend invece estende la lista se l'oggetto passato è un iterable altrimenti lancia un'eccezione.

La insert invece inserisce un elemento alla posizione specificata dall'utente, la posizione può essere anche negativa.

La remove rimuove l'elemento specificato come input, se questo non viene trovato viene lanciata un'eccezione.

Pop rimuove un elemento all'indice specificato e lo restituisce, funziona anche per indici negativi, se l'indice specificato non è valido lancia un'eccezione.

Clear rimuove tutti gli elementi dalla lista, ponendo a none gli attributi contenuti nei nodi, a partire da testa fino a coda.

La index fornito l'indice ne restituisce l'elemento, allo stesso modo dei precedenti metodi funziona anche per indici negativi e restituisce un'eccezione qualora non è stato trovato l'elemento.

Count conta le volte che un elemento compare nella lista, qualora non viene stato trovato restituisce 0.

Il metodo sort, ordina la lista e da documentazione di Python accetta una key, che è una funzione che permette di estrarre dalla classe dell'elemento l'attributo che è necessario per il paragone. Inoltre, accetta un parametro flag, che dice se oltre a ordinare la lista deve anche essere invertita.

Nel codice consegnato sono presenti diversi algoritmi di ordinamento, noi, date le assunzioni che l'algoritmo sarà usato su input molto grossi abbiamo scelto di usare il merge sort.

Il metodo reverse invece semplicemente scambia testa e coda e setta il flag _reverse a True.

Il metodo copy ritorna una shallow copy dell'oggetto in questione.

Abbiamo implementato gli operatori add e iadd, dove per iadd si estende la lista originale mentre in add se ne crea una nuova.

I metodi di confronto lanciano anch'essi un'eccezione se si prova a confrontare una lista con un tipo diverso.

Implementando len e __getitem__ Python ci ha automaticamente generato l'iteratore, e anche la funzione built-in bool.

Della lancia un'eccezione qualora non esista l'elemento a un indice specificato.

Eseguendo il modulo verifica vengono lanciati tutti i test che abbiamo realizzato sul nostro codice al fine di verificare l'ottenimento di un comportamento simile a quello della classe ufficiale.

Di seguito vengono riportate due tabelle, una indicante complessità temporale per ogni metodo/funzione richiesta dalla traccia e un'altra indicante i tempi di esecuzione richiesti per i test (considerando che in alcuni test abbiamo liste di più di 10000 elementi) sia con l'algoritmo di ordinamento merge sort che insertion sort.

Tabella complessità temporale

Metodo/funzione	Caso peggiore	Caso medio	Caso migliore
append(x)	$O(1)$	$O(1)$	$O(1)$
extend(iterable)	$O(n_e)$	$O(n_e)$	$O(1)$
insert(i,x)	$O(n)$	$O(n)$	$O(1)$
remove(x)	$O(n)$	$O(n)$	$O(1)$
pop([i])	$O(n)$	$O(n)$	$O(1)$
clear()	$O(n)$	$O(n)$	$O(1)$
index(x[, start[, end]])	$O(n)$	$O(n)$	$O(1)$
count(x)	$O(n)$	$O(n)$	$O(1)$

sort(key=None,reverse=False) Merge sort:	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Insertion sort:	$O(n^2)$	$O(n^2)$	$O(n)$
reverse()	$O(1)$	$O(1)$	$O(1)$
copy()	$O(n)$	$O(n)$	$O(1)$
+	$O(n_e)$	$O(n_n)$	$O(n_n)$
+=	$O(n_e)$	$O(n_e)$	$O(1)$
!=,==, <,>, >=, <=	$O(n)$	$O(n_{mylist2})$	$O(1)$
is	$O(n)$	$O(n)$	$O(1)$
my_list[i]	$O(n)$	$O(n)$	$O(1)$
my_list[i] = v	$O(n)$	$O(n)$	$O(1)$
del my_list[i]	$O(n)$	$O(n)$	$O(1)$
del my_list	$O(n)$	$O(n)$	$O(1)$
len	$O(1)$	$O(1)$	$O(1)$
bool	$O(1)$	$O(1)$	$O(1)$
str	$O(n)$	$O(n)$	$O(1)$

Tabella tempi di esecuzione:

Nome funzione/metodo	Conteggio chiamate	Tempo (ms)	Tempo dovuto alla propria computazione (ms)
ssr	3	0	0
ssi	5	0	0
sort	3	1636	0
reverse	3	0	0
remove	7	0	0
release	5	0	0
randrange	11505	36	16
random_string	123	3	0
randint	11505	42	6
rand_words_list	3	3	0
rand_keys_list	2	0	0
rand_dict	1	0	0
randList	2	47	6

pop	268600	629	192
insert	36	0	0
index	13	0	0
extend	13	11	1
count	2	0	0
copy	35	0	0
clear	4	0	0
append	278881	299	245
_merge_sort	20205	1636	120
_insertion_sort	30	328220	2890
_merge	10101	970	233
__setitem__	10	0	0
__ne__	2	0	0
__lt__	2	0	0
__len__	464861	42	42
__le__	2	0	0
__iadd__	2	0	0
__gt__	2	0	0
__getitem__	242176	148	148
__ge__	2	0	0
__eq__	4	0	0
__delitem__	10178	20	6
__del__	20258	34	14
__contains__	2	0	0
__add__	4	0	0

Dalla tabella, notiamo che c’è una certa differenza di tempo tra l’usare merge_sort o insertion_sort.