Università degli Studi di Salerno

Corso di Laurea Magistrale in Ingegneria Informatica

Tecniche di Programmazione

Documentazione Progretto n.1 Gruppo 14

Autori:
Giovanni Iodice
Francesco Liguori
Christian Rapido
Marco Ruggiero

Supervisori:
Prof. Vincenzo Auletta
Prof. Diodato Ferraioli

October 23, 2018

Indice

1 Introduzione

Obiettivo della prima parte del progetto è stato realizzare una classe, **CircularPositionalList**, che implementasse una lista rappresentata internamente con una Positional List circolare.

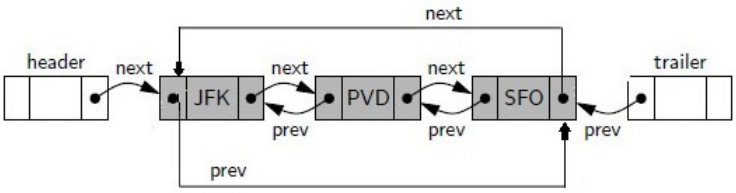

Figura 1: Struttura Lista Circolare

Si è quindi dovuto realizzare una struttura dati come quella in figura in cui:

- ciascun nodo, contenente un determinato elemento, è collegato al successivo e al precedente. In particolare, la proprietà di circolarità viene conseguita facendo in modo tale che il primo nodo della lista abbia per precedente l'ultimo, che, a sua volta, ha come successivo proprio il primo nodo;

- si ricorre a dei nodi *sentinella*, l'*header* e il *trailer* , che non contengono alcun elemento. L'header ha come successivo il primo nodo effettivo della lista, ma non ha alcun nodo come precedente. Il trailer, invece, non ha alcun elemento come successivo ma è preceduto dall'ultimo nodo effettivo della lista. La presenza di questi due nodi porta a notevoli semplificazioni nell'implementazione delle operazioni che verranno effettuate sulla lista. Innanzitutto, consente di trattare la lista vuota come una lista in cui compaiano solo header e trailer, con l'header che viene seguito dal trailer e il trailer che ha per precedente l'header. Ciascuna operazione di inserimento potrà allora essere sempre vista come un inserimento tra due nodi. Lo stesso di può dire per le cancellazioni. Anche l'accesso all'ultimo e al primo elemento della lista godono di una semplificazione a causa di header e trailer;

- ciascun nodo, tranne header e trailer, è contenuto in una Position i cui attributi saranno il nodo stesso e la lista che fa da *container* per tutte le

Position. Nascondere all'utente i nodi sottostanti inserendoli nelle Position permette di ottenre diversi vantaggi:

- l'utente non deve preoccuparsi di dettagli implementativi come il dover manipolare collegamenti tra i nodi o la presenza dei nodi sentinella e quindi c'è un incremento della **comprensibilità**;
- la struttura riceve un aumento in termini di **robustezza** proprio perché l'utente non può accedere e manipolare in maniere diretta i nodi;
- incapsulare i dettagli implementativi aumenta la **modificabilità** e la **flessibilità** della struttura dati che potrà essere più facilmente **riutilizzata** per diverse applicazioni.

La seconda parte del progetto ha previsto l'elaborazione delle informazioni contenute nella struttura dati circolare posizionale. E' stato infatti scritto un generatore, che sfruttando l'algoritmo di Bubble Sort, ordina e restituisce, gli elementi della lista. Invece, la funzione merge ha il compito di fondere due CircularPositionalList ordinate in una nuova CircularPositionalList, anch'essa ordinata.

La classe *ScoreBoard* è una forma di flessibilità e riusabilità di una CircularPositionalList. Uno ScoreBoard memorizza un certo numero di risultati di un gioco inserendoli all'interno lista circolare posizionale. A questo punto ogni Position conterrà per nodo uno Score, rappresentato da un nome, un punteggio e una data. In questo modo, tutte le funzionalità di uno ScoreBoard saranno implementate sfruttando i metodi della CircularPositionalList.

Il software prodotto è stato infine sottoposto ad una fase di testing. Questa ha avuto il compito di individuare errori e bug all'interno del progetto.

2 Implementazione

2.1 Classe CircularPositionalList

La classe CircularPositionalList estende la classe PositionalList. Da essa, eredita la struttura di tipo posizionale descritta precedentemente e i vari metodi. Inoltre, ogni metodo che riceve una Position in ingresso deve occuparsi di validarla tramite l'utility privata _validate(p) (complessità O(1)). Quest'ultima lancia due eccezioni:

- ValueError se p non appartiene al container o se non è considerata valida (p._node._next is None);
- TypeError se p non è un'istanza della classe Position.

Nel caso la validazione vada a buon fine, viene restituito il nodo contenuto nella position.

2.1.1 Metodi di accesso

first() Restituisce la Position dell'elemento che è identificato come il primo oppure None se la lista è vuota. Questo metodo viene ereditato, senza essere sovrascritto, dalla classe PositionalList. La sua implementazione consiste quindi nel restituire la Position dell'elemento successivo all'header. La complessità di first() è pertanto O(1).

last() Restituisce la Position dell'elemento che è identificato come l'ultimo oppure None se la lista è vuota. Anche last() viene ereditato, senza essere sovrascritto, dalla classe paterna. La sua implementazione consiste allora nel restituire la Position dell'elemento che precede il trailer. Al pari di first(), last() ha una complessità O(1).

before(p) Restituisce l'elemento nella Position precedente a p, None se p non ha un predecessore e ValueError se p non è una position della lista. Quindi, nel caso in cui la lista abbia dimensione unitaria, viene ritornato None. Viceversa, si sfrutta il metodo before(p) della classe paterna per trovare la Position dell'elemento precedente p. A questo punto, grazie al metodo element() sulla Position trovata è possibile ritornare l'elemento. La complessità è O(1).

after(p) Restituisce l'elemento nella Position successiva a p, None se p non ha un successore e ValueError se p non è una position della lista. Quindi, nel caso in cui la lista abbia dimensione unitaria, viene ritornato None. Viceversa, si sfrutta il metodo after(p) della classe paterna per trovare la Position dell'elemento successivo a p. A questo punto, grazie al metodo element() sulla Position trovata è possibile ritornare l'elemento. La complessità è O(1).

is_empty() Restituisce True se la lista è vuota e False altrimenti. Questo metodo verifica se la lunghezza della lista è nulla. In tal caso si ritorna True, altrimenti la lista non è vuota e viene ritornato False. Viene ereditato, senza essere sovrascritto. La complessità è O(1).

is_sorted() Restituisce True se la lista è ordinata e False altrimenti. Alla base di questo metodo c'è l'idea di scorrere la lista verificando che l'elemento in una Position sia maggiore dell'elemento contenuto nella Position successiva. Non appena questo confronto fornisce esito negativo allora si ritorna False. Viceversa, si continua l'iterazione fino a ritornare True. La complessità di quest'operazione è dovuta al ciclare su tutti gli elementi. Pertanto, la complessità è O(n).

find(e) Restituisce una Position contenente la prima occorrenza dell'elemento e nella lista o None se e non è presente. In particolare, si occupa di scorrere tutta la lista fino a quando non trova l'elemento da ricercare, e nel caso di insuccesso ritorna None. A causa del caso peggiore, elemento non trovato, la sua complessità è O(n), con n il numero di elementi presenti nella lista.

count(e) Restuisce il numero di occorrenze di e nella lista. Analizzando elemento per elemento della lista, tiene traccia di quante volte l'elemento e compare nella lista, aggiornando un opportuno contatore. Dovendo visitare tutta la lista la sua complessità computazionale è O(n), con n numero degli elementi.

2.1.2 Mutators

add_first(e) Inserisce l'elemento e in testa alla lista e restituisce la Position del nuovo elemento. Innanzitutto il metodo prevede l'inserimento di e in testa tramite l'add_first(e) della classe paterna, che si occupa anche di aumentare la dimensione della lista. Successivamente vengono modificati dei link in modo tale da garantire la circolarità. In particolare:

- l'ultimo elemento avrà come successivo il nuovo primo elemento della lista;
- il nuovo primo elemento avrà per precedente l'ultimo.

A questo punto la Position viene ritornata. Questo metodo si occupa di modificare un numero costante di link e quindi il costo è O(1).

add_last(e) Inserisce l'elemento e in coda alla lista e restituisce la Position del nuovo elemento. Ha un comportamento duale rispetto all'inserimento come primo elemento. Allo stesso modo è basato sul proprio corrispettivo paterno ma deve poi modificare dei collegamenti per garantire la circolarità:

- l'elemento aggiunto come ultimo avrà per successivo il primo;
- il primo elemento avrà per precedente quello appena inserito come ultimo.

Anche in questo caso, come prima, la complessità è O(1).

add_after(p,e) Inserisce un nuovo elemento e dopo il nodo nella Position p e restituisce la Position del nuovo elemento. Nel caso in cui la Position dopo cui bisogna effettuare l'inserimento sia l'ultima allora si inserisce l'elemento come ultimo usando add_last(e). Viceversa, se si deve inserire dopo una Position che non sia l'ultima, allora si ricorre al metodo della classe paterna. Anche il metodo add_after(p,e), dovendo solo modificare un numero costante di link, ha costo O(1).

add_before(p,e) Inserisce un nuovo elemento e prima del nodo nella Position p e restituisce la Position del nuovo elemento. E' questo il metodo duale di add_after(p,e). Se la Position prima della quale si deve effettuare l'inserimento è la prima allora si inserisce l'elemento come primo usando add_first(e). Viceversa, se si deve inserire prima una Position che non sia la prima, allora si ricorre al metodo della classe paterna. La complessità di questo metodo è O(1).

replace(p,e) Sostituisce l'elemento in Position p con e restituendo il vecchio elemento. Il suo compito è quello di sostuire un elemento già presente con uno nuovo: tale comportamento non cambia se la lista in uso è circolare. Pertanto, questo metodo è ereditato e non sovrascritto dalla classe padre. Avendo conoscenza della Position dell'elemento da sostituire, la complessità computazionale è di O(1).

delete(p) Rimuove e restituisce l'elemento in Position p dalla lista e invalida p. Questo metodo è basato sul riutilizzo del metodo delete(p) della classe paterna. In particolare, la cancellazione della Position avviene sempre tramite il metodo della classe paterna, ma è necessario determinare alcuni casi:

- la lista ha dimensione unitaria. Una volta effettuata la cancellazione, prima di ritornare l'elemento si ripristina la condizione di lista vuota di PositionalList e cioè i collegamenti di header e trailer in modo tale che l'elemento successivo dell'header sia il trailer e l'elemento precedente il trailer sia l'header;

- si cancella l'ultimo elemento ma la dimensione della lista non è unitaria. Effettuata la cancellazione, occorre ripristinare il campo trailer._prev in modo che punti correttamente all'elemento che in seguito alla cancellazione è diventato l'ultimo della lista. La funzione delete(p) della classe paterna si occupa infatti, data una Position p, di collegare tra loro l'elemento precedente e succesivo di p. A causa della circolarità della lista, tuttavia, l'ultimo elemento ha come successivo il primo. Allora, una volta effettuata la cancellazione occorre aggiornare trailer._prev che altrimenti punterebbe ancora all'elemento cancellato e quindi varrebbe None;

- si cancella il primo elemento ma la dimensione della lista non è unitaria. E' questo il caso duale del precedente: prima di ritornare e, è necessario aggiornare header._next per farlo puntare a quello che è diventato il primo elemento della lista in seguito alla cancellazione;

- si cancella un elemento in posizione intermedia. In questo caso si ricorre al metodo della classe paterna e si ritorna l'elemento cancellato.

In tutti i casi la delete prevede di modificare un numero costante di collegamenti e quindi la complessità è O(1).

clear() Rimuove tutti gli elementi della lista invalidando le corrispondenti Position. Si occupa di richiamare il metodo delete(p) per ogni elemento, pertanto la sua complessità è n volte il costo del metodo delete(p), quindi O(n), con n numero degli elementi della lista.

reverse() Inverte l'ordine degli elementi nella lista. Sfruttando l'implementazione dei nodi della lista, si è deciso di lasciare invariata la struttura dei

collegamenti tra i vari nodi e di scambiarne il contenuto. Pertanto il primo elemento verrà scambiato con l'ultimo, il secondo con il penultimo e così via. Tale funzione ricorre al metodo _swap(a, b) privato nella classe CircularPsitionalList, che si occupa dell'effettivo scambio del contenuto di due Position. La funzione si occupa di richiamare $\frac{n}{2}$ volte il metodo di swap che, a sua volta, invoca il metodo replace, il quale ha complessità O(1); la funzione reverse ha complessità quindi O(n), con n numero degli elementi.

copy() Restituisce una nuova CircularPositionalList che contiene gli stessi elementi della lista corrente memorizzati nello stesso ordine. In particolare crea una nuova CircularPositionalList e si occupa di aggiungere ad essa, in coda, via via una copia degli elementi della lista da replicare. Aggiungere un elemento in coda ha costo O(1), e iterando su tutti gli elementi della lista, porta la complessità a O(n), con n numero degli elementi della lista.

2.1.3 Magic method

__add__() Crea una nuova lista con tutti gli elementi di x e tutti gli elementi di y, inseriti dopo l'ultimo elemento di x. Viene utilizzato come magic method di x + y. La complessità è O(n+m), dove n è il numero degli elementi della prima lista, m il numero di elementi della seconda lista.

__contains__() Restituisce True se p è presente nella lista e False altrimenti. Usato con la sintassi p in x, verifica che p sia una Position valida e appartente alla CircularPositionalList x. La sua implementazione richiama il metodo _validate() catturandone l'eventuale eccezione, pertanto ha complessità O(1).

__getitem__() Usando la sintassi x[p] restituisce l'elemento contenuto nella position p. Servendosi del metodo find() per ricercare l'elemento, l'operazione costa complessivamente O(n), con n numero degli elementi della lista.

__setitem__() Sostituisce l'elemento nella position p con e, servendosi la sintassi x[p] = e. Dovendo invocare la replace ha un costo di O(1).

__len__() Restituisce il numero di elementi contenuti in x. Invocato con len(x), viene ereditato senza essere sovrascritto ed ha costo O(1).

__delitem__(key) Rimuove l'elemento nella position p invalidando la position, il metodo è invocato usando la sintassi del x[p]. L'operazione ha una complessità di O(1).

__iter__() Generatore che restituisce gli elementi della lista a partire da quello che è identificato come primo fino a quello che è identificato come ultimo. Dovendo scorrere tutta la lista, la complessità è di O(n), con n numero degli elementi.

__str__() Rappresenta il contenuto della lista come una sequenza di elementi, separati da virgole, partendo da quello che è identificato come primo. Dovendo fornire una rappresentazione per ogni elemento della lista, ha un complessità di O(n), con n numero degli elementi.

2.1.4 Metodi privati di supporto

_swap(first_obj, last_obj) è stato creato come metodo di supporto per il metodo reverse. Esso non fa altro che scambiare gli elementi tra le posizioni passate come parametri. La sua complessità è dunque O(1).

_get_prev(p) ritorna la position dell'elemento precedente a p o None se non ha un predecessore. Viene richiamato il metodo before della classe padre e pertanto, ha complessità O(1).

_get_next(p) ritorna la position dell'elemento successivo a p o None se non ha un successore. Viene richiamato il metodo after della classe padre e pertanto, ha complessità O(1).

2.2 Generatore bubblesorted()

È un generatore che ordina gli elementi della lista e li restituisce nell'ordine crescente. Il generatore non modifica l'ordine in cui sono memorizzati gli elementi della lista. Si crea una copia della lista corrente e su questa si procede ad effettuare l'ordinamento tramite l'algoritmo Bubble Sort. Il generatore ha una complessità di O(n^2).
Esso assume inoltre che gli elementi della lista siano confrontabili tra loro. Viceversa, verrebbe lanciata un'eccezione.

2.3 Funzione Merge(l1,l2)

Prende in ingresso due CircularPositionalList, già ordinate, e provvede a crearne una nuova inserendo gli elementi di entrambe le liste in modo da ottenerla ordinata. Per fare ciò c'è bisogno di scorrere tutti gli elementi delle due liste provvedendo, ad effettuare un confronto elemento per elemento e ad inserire, nel nostro caso, l'elemento con valore maggiore. Pertanto la complessità è O(n+m), dove n è il numeri degli elementi della prima lista, m il numero di elementi della seconda.

2.4 Classe ScoreBoard

La classe ScoreBoard permette di salvare una lista di "x" risultati. I risultati sono stati implementati attraverso l'utilizzo di una classe, denominata Score, definita internamente a Scoreboard. In particolare, ogni istanza di Score avrà gli attributi :

- Nome
- Score
- Data

Che verranno inizializzati grazie al costruttore. È risultato necessario andare a sovrascrivere il comportamento di determinati operatori come less-than e greater-than. Tali operatori stabiliscono un confronto tra oggetti della classe Score, in particolare riflettono l'ordine naturale del campo score. L'implementazione ha complessità O(1). Inoltre è stato sovrascritto anche l'operatore __str__(), per poter stampare un oggetto Score. Viene rappresentato attraverso una stringa che ne riporta gli attributi separandoli con un trattino. Tale metodo ha complessità O(1). Infine, per la classe Score sono stati definiti i metodi di getter, per valutare le informazioni contenute nei campi di un'istanza della classe. In particolare, si usano score(), nome(), date(), per accedere rispettivamente a _score, _nome, _date.
Tornando alla classe Scoreboard essa avrà come attributi una CircularPositionList e un campo dimension per indicare il numero massimo di score inseribili.Tale campo viene controllato se è intero positivo. Per indicare il numero di elementi presenti all'interno, restituito dal metodo size(), non è stato inserito un ulteriore attributo ma è stata utilizzata la lunghezza della CircularPositionList. Tale campo è utilizzato anche dal metodo is_empty().
Si è deciso di organizzare lo ScoreBoard in maniera tale da essere ordinato ed avere gli score più elevati nelle posizioni di testa come, ad esempio, nelle classifiche sportive.

2.4.1 Inserimento

Prima di effettuare l'inserimento controlliamo se lo score che si vuole aggiungere è effettivamente un'istanza di Score e se gli attributi sono corretti. Tale controllo avviene tramite il metodo privato denominato _score_validate(score).
Viene controllato anche se lo score è già presente o meno all'interno della scoreboard richiamando il metodo privato _score_equals(score2) e, se presente non viene reinserito confermando a video quanto accaduto.
Se non presente, si determina attraverso un ciclo while qual è l'elemento che precede o succede lo score che deve essere inserito. Se l'inserimento provoca un eccesso rispetto alla dimensione dello ScoreBoard, l'ultimo elemento che corrisponde allo score peggiore, viene eliminato.
La complessità è O(n) derivante dal ciclo while che, nel caso peggiore, deve scorrere tutti gli elementi e dal controllo sull'eventuale già presenza dello score.

2.4.2 Merge(new)

Per l'implementazione di Merge(new), si è deciso di utilizzare il seguente approccio. Andiamo ad unire entrambi gli ScoreBoard indipendentemente dal numero di elementi salvandoli in una CircularPositionalList di appoggio. Dopodichè si va a contare il numero di elementi che risultano in eccesso rispetto alla dimensione dello ScoreBoard su cui la Merge(new) è stata richiamata. Dato che lo ScoreBoard presenta in testa gli score più elevati, quelli da eliminare saranno nelle ultime posizioni ed eliminati uno alla volta.
Fatto ciò, gli elementi rimanenti verranno inseriti nello ScoreBoard corrente e la CircularPositionalList di appoggio distrutta.
La complessità di questo algoritmo è pari a O(n+m), cioè la complessità della funzione merge richiamata.

2.4.3 Top e Last

Top(i=1) Il metodo Top restituisce i primi i elementi e se non settato, di default restituisce soltanto il primo. Un primo controllo viene effettuato proprio su i per verificare che esso non ecceda la dimensione dello scoreboard e che sia positivo, altrimenti lancia un'eccezione di tipo ValueError. Se risulta nel range, allora, si punta al primo elemento e si procede restituendo gli i elementi richiesti in una lista.
La complessità computazionale è pari a O(i), con i che vale n nel caso peggiore quando cioè bisogna scorrere l'intero Scoreboard.

Last(i=1) Il metodo Last restituisce gli ultimi i elementi e se non settato, di default ne restituisce soltanto uno. Anche qui come per Top, viene effettuato un controllo sul valore di i. Dopodichè punto all'ultimo elemento e scorro in ordine inverso lo scoreboard restituendo gli i elementi richiesti in una lista.
La complessità è O(i) per gli stessi motivi elencati per top().

2.4.4 Metodi privati di supporto

_score_validate(score) effettua un controllo sulla validità dello score e sugli attributi che lo compongono, allo stesso modo del metodo _validate appartenente alla classe PositionalList. La sua complessità è O(1).

_score_equal(score) esegue un ciclo in cui controlla elemento per elemento se, uno score presente è uguale a quello passato come parametro. Dovendo attraversare tutta la lista, tale metodo ha complessità O(n).