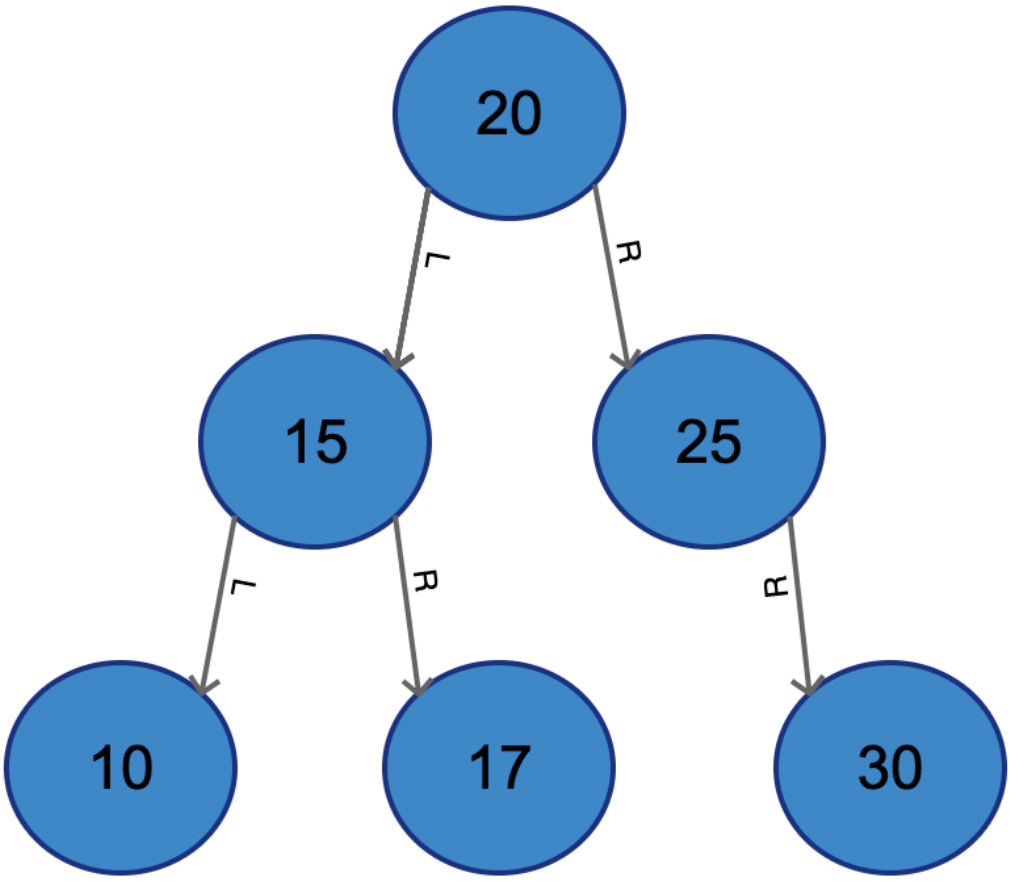
816390
Lorenzo Gallorini
l.gallorini1@campus.unimib.it
Programmazione C++
29 marzo 2020

Esame di Programmazione C++



Progetto C++

Descrizione del progetto

Il progetto consiste nella realizzazione di una classe generica che implementa un albero binario di ricerca. L'albero è formato da un insieme di elementi T contenuti in nodi connessi in una struttura generica padre<—>figlio e **NON** deve permettere l'inserimento di dati duplicati. Deve essere possibile per l'utente scegliere la strategia per confrontare due dati T.

Ho creato quindi una classe ABR<T,C,D> dove T è il tipo di dato che accetta l'albero; C è il modo con cui confronto se una dato è **strettamente maggiore** o **strettamente minore** di un altro, poiché **NON** permetto l'inserimento di dati duplicati; D lo utilizzo per identificare se il valore di cui sto cercando il nodo è uguale al valore di un nodo.

Per la gestione dei Nodi dell'albero ho creato una struct di supporto chiamata Nodo, la quale contiene il Valore del nodo, il puntatore al figlio sinistro, il puntatore al figlio destro e il puntatore al padre. Ho scelto di tenere un puntatore al padre per rendere più semplice l'implementazione dell'iteratore, il quale ripercorre facilmente verso l'alto l'albero partendo del nodo più a sinistra[1].

Commento all'implementazione

Come metodi fondamentali della classe ABR<T,C,D> ho implementato il costruttore, il distruttore, un metodo che aggiunge un Nodo all'albero, un metodo che rimuove un nodo dall'albero e un metodo che restituisce il valore nel nodo più a sinistra (vedi nota 1).

Come da specifiche precedentemente fornite nella classe sono implementati i seguenti punti:

1. Il numero totale di dati inseriti nell'albero;
   ```
   int Count();
   ```
   Il metodo restituisce il numero di elementi contenuti nell'albero.
   Per contare gli elementi sfrutto un algoritmo ricorsivo per visitare l'albero:
   ```
   int CountR(const Nodo *SubRoot);
   ```

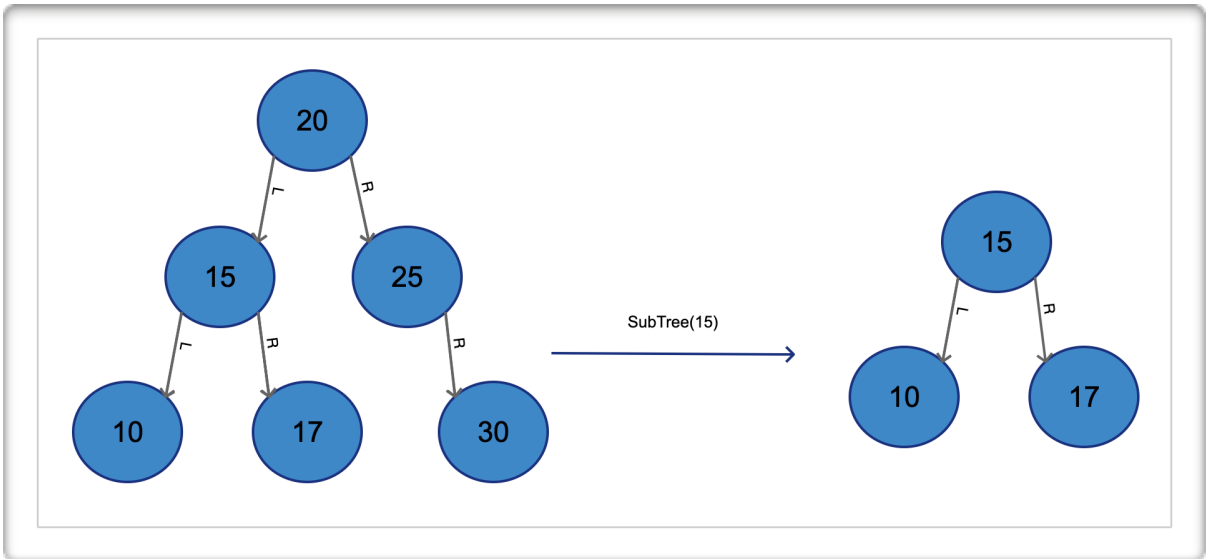
2. Il controllo di esistenza di un elemento di tipo T.
   ```
   bool Find(const T &Value);
   ```
   Iterativamente vado a controllare tutto l'albero sfruttando il suo ordinamento.

[1] Solitamente l'albero binario per definizione sfrutta l'operatore di confronto <, quindi in un albero binario "classico" il posto più a sinistra corrisponde al valore più piccolo dell'albero.

3. Di accedere ai dati presenti nell'albero tramite un iteratore a sola lettura e di tipo forward. L'ordine con il quale sono ritornati i dati non è rilevante.
   Ho creato una classe const_iterator dove ho ridefinito i vari operatori in base alle mie esigenze. La modifica più rilevante è quella dei metodi degli incrementatori (pre-incremento e post-incremento), l'algoritmo è molto semplice: partendo dal nodo più a sinistra, se si può andare a destra si procede andando in quella direzione, poi andando tutto a sinistra, altrimenti ripercorro i padri finché non arrivo da dove sono arrivato a sinistra.

4. Di stampare il contenuto dell'albero (anche usando operator<<).
   Per la stampa dell'albero ho ridefinito lo standard output in modo da accettare un oggetto di tipo ABR e stamparne il suo contenuto. Tutto questo è stato fatto seguendo una logica in-Order, sfruttando come appoggio l'iteratore definito precedentemente.

5. Implementare, inoltre, un metodo SubTree che, passato un dato T dello stesso tipo del dato contenuto nell'albero, ritorna un nuovo albero. Il nuovo albero deve corrispondere al sottoalbero avente come radice il nodo con il valore passato come parametro.
   ABR SubTree(const T &Value);
   Questo metodo inizialmente l'avevo concepito in una maniera differente di come l'ho sviluppato nella versione finale. Inizialmente, il sottoalbero creato condivideva le stesse celle di memoria del vecchio albero, ma successivamente ho ritenuto che questa soluzione avesse poco senso e potesse creare molti problemi da gestire …
   Nella versione attuale, infatti, dopo aver creato un nuovo albero, vado ad aggiungergli tutti i nodi ricorsivamente utilizzando una logica pre-order (VLR), al fine di mantenere la stessa struttura dell'albero di partenza.

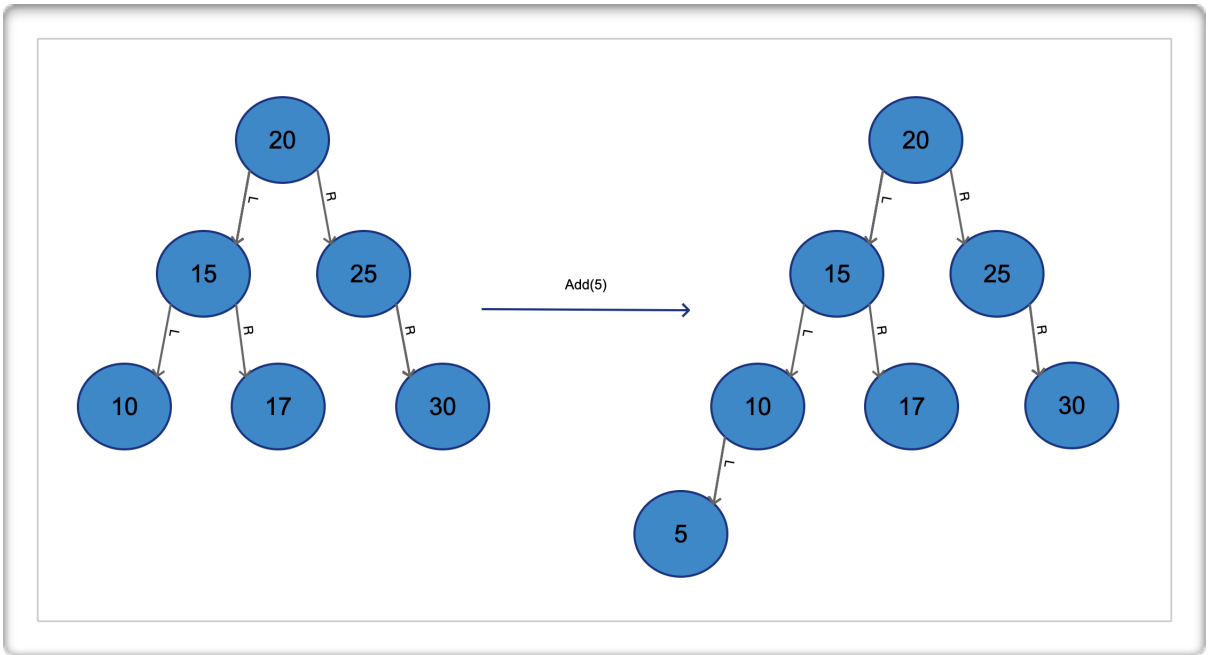


6. Implementare una funzione globale PrintIF che, dato un albero binario di tipo T e un predicato P, stampa a schermo tutti i valori contenuti nell'albero che soddisfano il predicato.
   ```
   void PrintIF(const ABR<T,C,D> &Albero,Predicato P);
   ```
   Questo metodo è concettualmente molto simile alla ridefinizione dello standard output. La differenza è che si tratta di un metodo globale che accetta in input un albero ABR e un predicato P. Successivamente stampa a video sullo standard output solo gli elementi dell'albero che soddisfano il predicato P.

Di seguito vado ad elencare le funzionalità che ho già stabilito come basiche da implementare all'interno della classe ABR<T,C,D>.

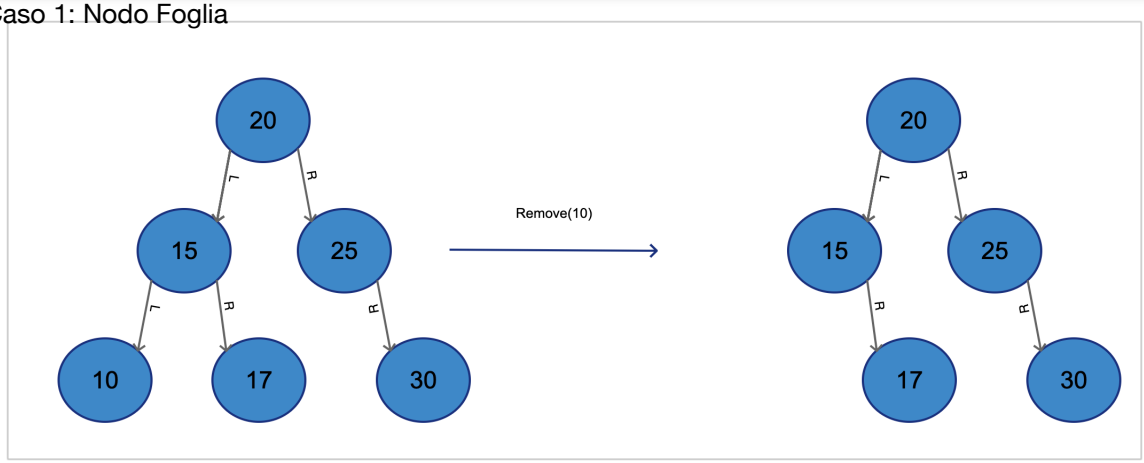
A. Costruttori.
   ```
   ABR() : _Root(nullptr) {}
   ABR(const T &value) : _Root(new Nodo(value)) {}
   ABR(const ABR &other) : _Root(nullptr) { PopolaAlbero(other._Root); }
   ```

B. Un metodo Add che aggiunge, in base all'ordine che abbiamo dato al nostro albero, il nodo nel punto corretto.
   ```
   void Add(const T &Value);
   ```
   Il metodo funziona in maniera iterativa andando a sinistra se il valore da aggiungere è "minore" del nodo e andando a destra se "maggiore". Ovviamente si può comportare in maniera contraria (o con una logica differente) se definito diversamente in fase di dichiarazione dell'albero.

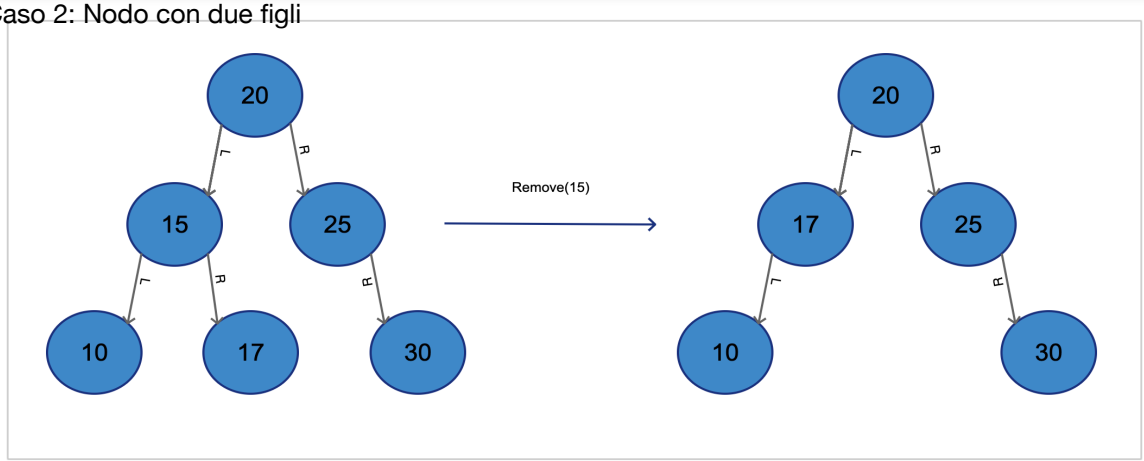


C. Un metodo Remove che elimina un nodo cercandolo tramite il suo valore.
`void Remove(const T &Value)`
La rimozione dei nodi è stata divisa in tre casi e nelle seguenti illustrazioni viene mostrato come vengono gestiti.

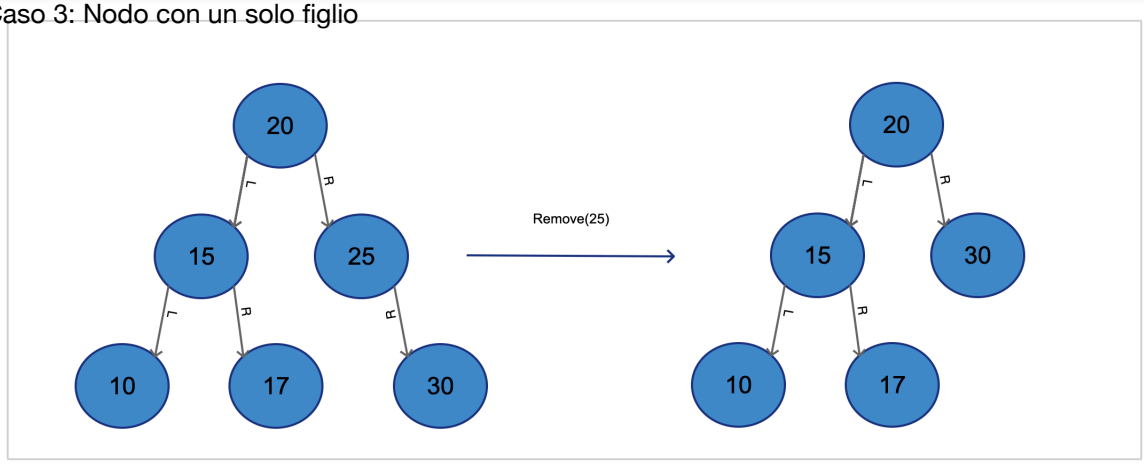
Caso 1: Nodo Foglia



Caso 2: Nodo con due figli



Caso 3: Nodo con un solo figlio



D. Ho ridefinito l'operatore di assegnamento (=) in modo da copiare il contenuto di un albero in uno nuovo.

E. L'ultimo metodo che ho implementato è il MinimumValue
   T MinimumValue();
   Questo metodo permette a chi utilizza la libreria di sfruttare la funzione privata ValoreMinimo che raggiunge il nodo più a sinistra dell'albero e restituisce il valore. Il nome MinimumValue nasce appunto dal fatto che solitamente in quel nodo è contenuto il valore minimo, poiché possiamo cambiare l'ordine degli elementi, non verrà restituito necessariamente il valore più basso, ma semplicemente quello più a sinistra.

Gestione degli errori

Ho deciso di implementare una semplice gestione degli errori, associando ad ogni tipo di errore un codice.
Se si tenta di inserire un elemento già inserito verrà lanciata una *Elemento_gia_inserito_exception*, la quale ha come *error_cod* -1. Se si cerca di fare operazione su un nodo inesistente verrà lanciata una *Elemento_non_trovato_exception* con *error_cod* -2. Se si tenta di fare operazioni non consentite su un albero non inizializzato viene lanciata una *Albero_non_inizializzato_Exception* con *error_cod* -3, infine se durante una new dovesse essere lanciata una eccezione la gestisco lanciando però a mia volta una eccezione chiamata New_Nodo_Bad_Alloc_Exception con error_cod -4.

Fase di Debug

Per testare l'affidabilità del mio codice ho deciso di effettuare un test basato sulla copertura degli statement. Il mio obbiettivo era quello di raggiungere uno statement coverage del 100% per essere sicuro di aver compilato tutte le parti del progetto.

Ho effettuato dei test anche per quanto riguarda la perdita di memoria (memory leak) utilizzando come strumento Valgrind su due sistemi operativi differenti MacOs Mojave e Linux Mint. Non ho riscontrato nessuna perdita di memoria in nessuno dei due sistemi operativi.

Progetto Qt

Descrizione del progetto

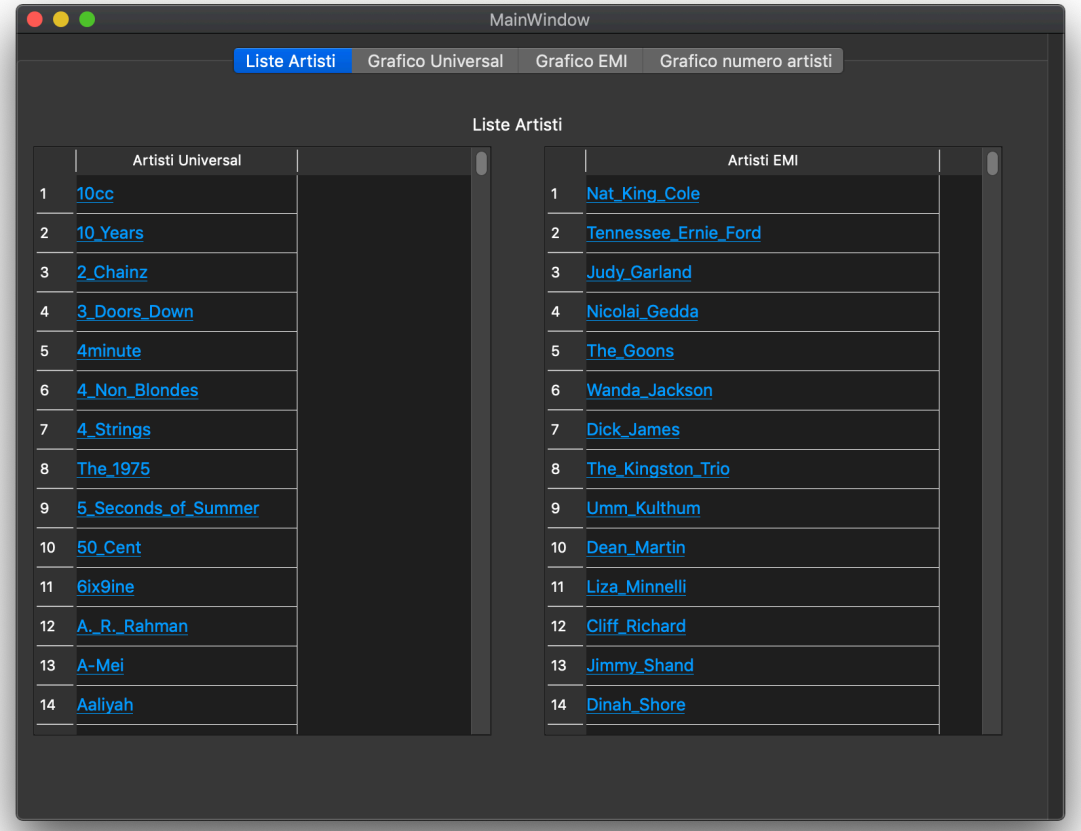
Il progetto richiede un'interfaccia grafica per la visualizzazione degli artisti di due delle più prestigiose etichette discografiche mondiali.
Per questo progetto ho utilizzato l'IDE QtCreator 4.11.1 nella sua ultima versione e la libreria Qt 5.14.1 anche lei l'ultima rilasciata.

Commento all'implementazione

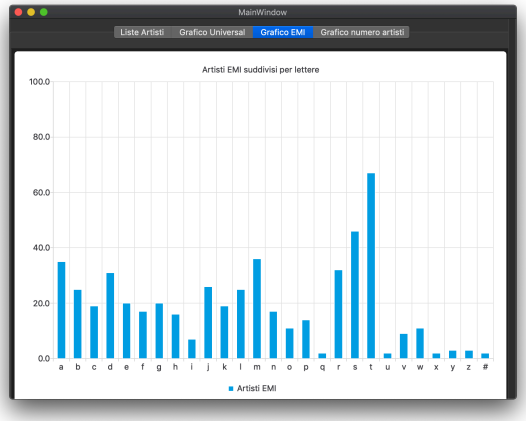
Ho svolto l'implementazione per punti proprio come indicati nel PDF delle specifiche.

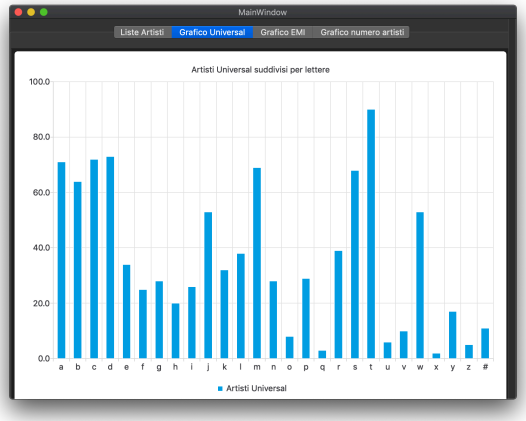
1. Per scaricare i due file relativi alle etichette mi connetto al server e recupero il file di testo sotto forma di stringa senza scaricare esplicitamente il file.
   Metodo di riferimento: QString GetFromUrl(QString url);
2. Per creare la lista di artisti da visualizzare, lavoro sulla stringa creando poi una lista di stringhe dove ogni cella contiene il link alla pagina wikipedia.
   Metodo di riferimento: QStringList LinkedList(QString stringa);

3. La lista creata con il metodo precedente viene passata alla classe MainWindow per visualizzare i dati a schermo, tramite delle QLabel posizionate all'interno di una Tabella.
   Ho deciso di creare un metodo per ogni Tabella per evitare problemi di qualunque tipo.
   Metodi di riferimento: void MainWindow::setListEMI(QStringList lista);
   void MainWindow::setListUNIVERSAL(QStringList lista)

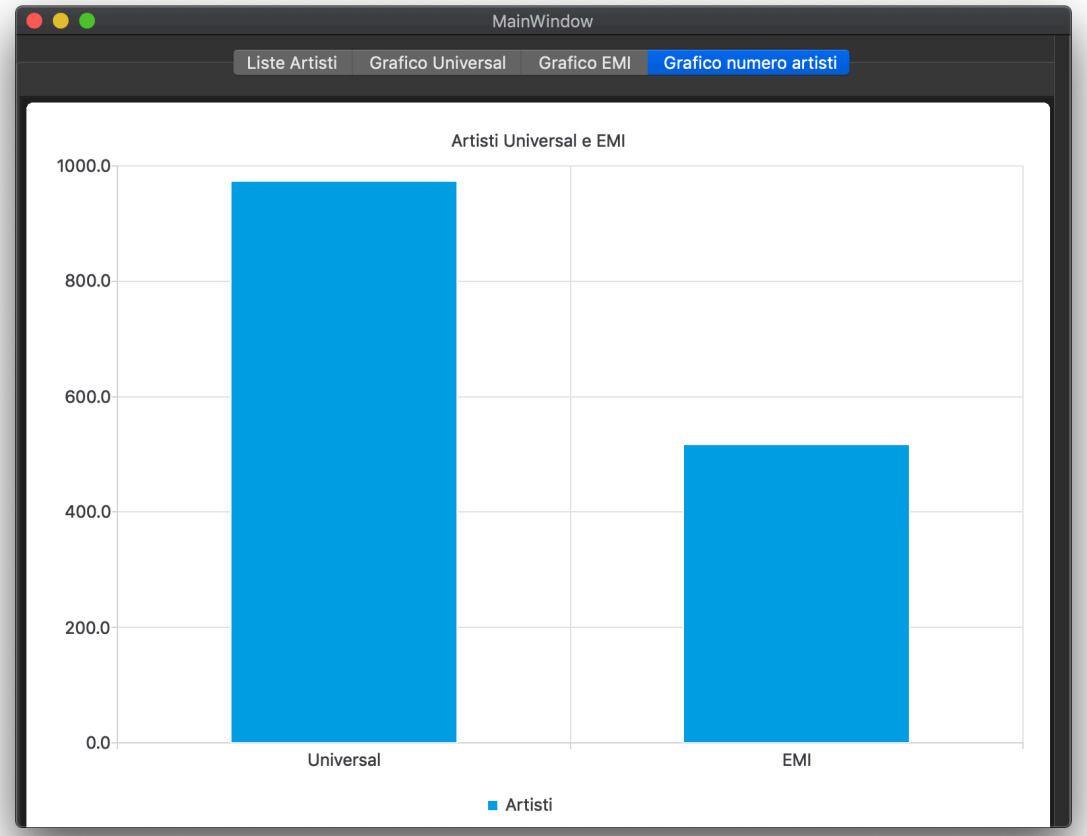


4. La creazione dei grafici raggruppati per iniziale dell'artista richiedeva prima il conteggio delle iniziali e successivamente il rendering a video dei risultati.
   Metodi di riferimento: void CountLetter(QString stringa, bool isEMI);
   void MainWindow::renderGraphic(int array[27],bool isEMI)





5. L'ultimo grafico è stato relativamente semplice da ottenere: ho riciclato il codice del grafico precedente e come set di dati ho usato il numero di elementi nelle liste generate in precedenza. Metodo di riferimento: void MainWindow::renderGraphic2(int Universal,int EMI).



Fase di Debug

Un debug dal punto di vista degli Statement non aveva molto senso in questo caso, dato che il programma è stato costruito per eseguire ogni riga di codice. Ho cercato più che altro di constatare che il programma soddisfasse le varie specifiche.
Dal punto di vista dei Memory Leak in ambiente MacOs Mojave ho avuto dei problemi ad eseguire i test con Valgrind e non sono riuscito a testarlo in questo ambiente, mentre in ambiente Linux Mint tramite Qt Creator sono riuscito a fare il test per quanto riguarda la perdita della memoria non riscontrando problemi.