RELAZIONE PROGETTO

“Programmazione C++” | Giugno 2024

Marcaccio Riccardo – 886023

r.marcaccio@campus.unimib.it

C++

Classe

Ho implementato la classe `Multiset` utilizzando una classe templata per memorizzare un insieme di elementi generici. La classe implementa tre costruttori, un distruttore e due ridefinizioni di operatori.

Costruttori

1. **Costruttore Base:**
   È stato implementato in modo che, richiamando il costruttore senza argomenti, la classe venga inizializzata mettendo il puntatore di testa a `nullptr`.
2. **Costruttore da Due Iteratori:**
   Questo costruttore utilizza due iteratori, uno di inizio e l'altro di fine, per scorrere la struttura e aggiungere gli elementi con la funzione interna `add(T val)`.
3. **Costruttore di Copia:**
   Implementato per prendere una reference all'oggetto da copiare e richiamare la ridefinizione del costruttore, evitando la ripetizione del codice.

Distruttore

Il distruttore è stato implementato attraverso un metodo privato *clear()*, che permette di rimuovere e deallocare tutti gli elementi presenti nel multiset. Dato che questa funzione è necessaria anche per l'assegnazione tra Multiset, ho deciso di implementare *clear()* piuttosto che ripetere il codice.

Ridefinizione degli Operatori

- **Operatore =:**
  L'operatore di assegnamento è stato ridefinito per permettere la copia di un oggetto già istanziato in modo rapido e semplice.
- **Operatore ==:**
  L'operatore è stato implementato per confrontare due oggetti Multiset, come richiesto.

Struttura

Per realizzare il progetto, ho utilizzato una lista a puntatori denominata *`Pair_node`* e dichiarata privata. Questa contiene due attributi:

Un oggetto *`Pair`*: utilizzato per memorizzare le coppie valore-occorrenze.
Un puntatore *`next`*: utilizzato per passare da un elemento al successivo nella lista.
Ho creato un attributo *`head`* di tipo puntatore a *`Pair_node`*, che identifica la testa della lista a puntatori. Questa struttura è dinamica, permettendo di aggiungere o rimuovere oggetti in qualsiasi punto della lista.

Per ritornare uno struct pubblico *`Pair`* tramite un iteratore, ho dichiarato lo struct separatamente e successivamente incluso nel nodo, isolando così il puntatore *`next`* dall'oggetto *`Pair`*. La struct pubblica *`Pair`* è stata implementata con un attributo costante di tipo T (tipo a cui è stata templata la classe) per registrare i valori e un intero per le occorrenze.

Gestione della Struttura

La struttura può essere modificata tramite aggiunta e rimozione di valori. I due metodi pubblici responsabili di queste operazioni sono *`add(T)`* e *`remove(T)`*. Per migliorare la chiarezza e la manutenibilità, ho separato la ricerca della posizione di inserimento/rimozione dall'effettiva esecuzione dell'azione richiesta.

Funzione *relative_pos(T)*

Ho implementato una funzione privata *relative_pos(T)* che restituisce un puntatore al nodo precedente rispetto al valore T. I possibili risultati di questa funzione sono tre:

1. Un puntatore a un nodo.
2. *nullptr* in caso di lista vuota o inserimento in testa.
3. *nullptr* in caso di valore non trovato.

Restituire il nodo precedente è necessario poiché, lavorando con una lista a puntatori, è necessario modificare l'elemento *next* del nodo in questione. L'ambiguità derivante dal ritorno di *nullptr* in vari contesti è risolta nel metodo di rimozione, mentre non è applicabile al metodo di inserimento. Nel caso della *remove()*, infatti, se *relative_pos* ritorna *nullptr*, è necessario verificare se il valore da rimuovere è contenuto nel nodo di testa per procedere con la rimozione.

Gestione delle Eccezioni

La gestione delle eccezioni segue il principio secondo cui ogni metodo è responsabile esclusivamente della propria logica specifica, delegando la gestione degli errori ai livelli superiori, dove il contesto è più chiaro. Analizzando i vari metodi, ritengo che l'unico metodo che potrebbe causare problemi sia il metodo *add()*.

Le eccezioni generate dal metodo *add()* vengono gestite nei metodi che lo utilizzano direttamente. Per garantire un utilizzo sicuro della classe, la documentazione specifica che il metodo *add()* potrebbe lanciare un'eccezione, raccomandando di gestirle appropriatamente. Questa strategia permette di mantenere i metodi semplici e focalizzati, mentre la gestione delle eccezioni avviene in un contesto dove è più semplice prendere decisioni su come procedere in caso di errore.

Metodi di Accesso

La classe Multiset offre vari metodi di accesso per interagire con i dati contenuti al suo interno. Questi includono:

- *`contains(T val)`*: Verifica la presenza di un valore.
- *`occurrences_by_value(T val)`*: Ottiene il numero di occorrenze di un valore specifico
- *`total_occurrences()`*: Calcola il numero totale di occorrenze di tutti i valori.
- *`total_values()`*: Ottiene il numero di valori unici presenti nel multiset.

Questi metodi permettono di interrogare efficacemente il multiset, fornendo informazioni dettagliate sui dati memorizzati.

QT

Classe

La classe mainwindow presenta due attributi privati da me aggiunti:

- *img_pixels* : unsigned int, contiene la dimensione in numero di pixel della immagine inserita. E' un parametro che utilizzo per normalizzare i vari grafici per il numero di pixels.
- *RgbMap* : Qmap <QRgba64, int>, contiene le coppie (colore, occorrenze).

Metodi ed Elementi grafici

Gli elementi grafici che sono stati utilizzati per realizzare il progetto sono i seguenti:

- Bottoni
  - InserisciIMG : Il tasto che utilizzo per caricare le immagini all'interno del programma.
  - RimuoviIMG : Il tasto per "resettare" lo stato del programma. Rimuove l'immagine e azzera la struttura dati.
  - IstogrammaRGB : Il tasto per la visualizzazione di un grafico.
- ComboBox : Elemento che utilizzo per permettere la scelta del grafico da visualizzare. Le possibili scelte sono "Grafico RGB", "Grafico R", "Grafico G", "Grafico B".
- label : Elemento che contiene l'immagine attraverso la sua proprieta` setPixmap().
- CheckBox : Elemento che permette all'utente di visualizzare o meno i codici colori con occorrenze a 0 per i grafici singolo canale.

Slots

Gli slot che utilizzato sono Button_clicked() e comboBox_currentIndexChanged().

Il primo slot permette ai bottoni d i richiamare la funzione corrispondente, in modo da portare a termine il loro compito.

Il secondo, viene utilizzato per abilitare e disabilitare la checkBox in base al tipo di grafico che si vuole visualizzare.

Struttura Dati

La struttura dati che ho scelto per salvare le intensita` RGB e` una Qmap con key un QRgba64 e value int. Grazie a questo tipo di struttura l'inserimento e il controllo sull'esistenza di una sola istanza di un colore e` stato molto piu` semplice. Inoltre la Qmap viene automaticamente ordinata per chiave, permettendo quindi di avere un grafico con dei colori in scala (dal piu` vicino a 0 al piu` lontano a 255).

Per salvare le intensita` dei singoli canali ho deciso di utilizzare la stessa struttura, in modo da avere una sola funzione di stampa. In questo modo, una volta lette le triplette RGB dall'immagine, e` sufficente creare una nuova struttura utilizzando solo il canale prescelto.