Relazione progetto C++

Giugno 2020

Nome: Alessandro

Cognome: Gallizzi

Matricola: 830104

E-mail: a.gallizzi@campus.unimib.it

Tempistiche: Dal 15/05/2020 al 05/06/2020

Introduzione

Dopo aver valutato il problema proposto, ho deciso di implementare una coda dinamica avente come singolo elemento una struct *element* contente come dati: un valore generico T, un puntatore all'elemento successivo e un puntatore all'elemento precedente. L'implementazione della coda dinamica permette quindi una facile gestione della struttura dati rappresentata, avendo il vantaggio di non dover stabilire a priori, staticamente, il numero di elementi che questa avrà al suo interno. Le operazioni di inserimento e rimozione sono eseguite in tempo costante O(1).

Tipi di dati

Il tipo di dato scelto è un elemento che, come specificato nella parte introduttiva, è una struct contenente tre dati:

- **Value** di tipo generico T. Contiene il valore che è stato assegnato ad un elemento della coda.
- **next** di tipo puntatore element. È il puntatore all'elemento successivo, questo vuol dire che punta, se esiste, all'elemento inserito successivamente all'elemento corrente nella coda.
- **prev** di tipo puntatore element. È il puntatore all'elemento precedente, questo vuol dire che punta, se esiste, all'elemento inserito precedentemente all'elemento corrente nella coda.

La struct *element* è dichiara nel file coda.h. L'unico costruttore inizializzato è quello in cui viene passato per parametro un valore generico di tipo T. È stata scelta questa implementazione perché sarebbe inutile la creazione di un elemento senza valore alcuno.

Eccezioni

empty_queue_exception: eccezione custom che viene generata quando si rimuove un elemento, oppure si utilizzano metodi di get e set su una coda vuota. È stata decisa l'implementazione di una eccezione custom per precisare il caso specifico in cui si commette un errore su una coda vuota.

Implementazione

La classe coda contiene come attributi:

- **_head** di tipo puntatore element. È il puntatore al primo elemento inserito nella coda.
- **_tail** di tipo puntatore element. È il puntatore all'ultimo elemento inserito nella coda.
- **_size** di tipo unsigned int. È la dimensione della coda, quindi il numero di elementi inseriti.

Si ha anche un funtore d'uguaglianza di tipo generico E **_equal** di supporto agli elementi della coda, in specifico al metodo search.

Oltre i fondamentali costruttore di default, costruttore di copia, distruttore e operatore d'assegnamento; è stato implementato un costruttore che permette l'inizializzazione della coda a partire da due iteratori.

Il costruttore di default inizializza la coda con la dimensione zero, quindi con _head e _tail nulli.

Il costruttore di copia utilizza un metodo privato *copy_helper* per costruire una coda a partire dalla _head della coda da copiare. In caso un elemento della coda da copiare risulti nullo, viene lanciata una allocazione di memoria.

L'operatore di assegnamento utilizza il metodo swap della classe algorithm.h per scambiare gli attributi della coda corrente con la coda di cui si vuol fare l'assegnamento. Viene ritornato un reference a this.

Il distruttore libera ogni allocazione di memoria grazie al metodo *clear*.

Metodi implementati

I metodi implementati sono i seguenti:

- **enqueue**: permette l'inserimento di un elemento dalla coda. Il parametro che viene passato è una costante di dato generico di tipo T. L'inserimento viene fatto alla fine della coda, questo per mantenere la logica implementativa della struttura dati che si andrà a generare. Viene generata un'eccezione di allocazione di memoria nel caso la costruzione dell'elemento da inserire non vada a buon fine;
- **dequeue**: permette la cancellazione di un elemento dalla coda. La cancellazione viene fatta all'inizio della coda, questo per mantenere la logica implementativa della struttura dati che si andrà a generare. Viene generata un'eccezione custom *empty_queue_exception* nel caso si stia cercando di cancellare un elemento da una coda vuota. Viene ritornato il valore generico T dell'elemento che è stato, eventualmente, cancellato;
- **clear**: permette la cancellazione di ogni elemento memorizzato nella coda e imposta la dimensione della coda a 0, per fare questo si serve del metodo **clear_helper** che, dato come parametro l'elemento _head procede ricorsivamente a cancellare tutti gli elementi dalla coda;
- **size**: ritorna la dimensione della coda accessibile dall'attributo privato _size;
- **print**: stampa ciclicamente i valori degli elementi presenti in una coda dal meno recente al più recente;

- **getHead**: ritorna il valore della _head della coda. Nel caso la coda non abbia elementi viene generata una eccezione custom *empty_queue_exception*;
- **getTail**: ritorna il valore della _tail della coda. Nel caso la coda non abbia elementi viene generata una eccezione custom *empty_queue_exception*;
- **setHead**: cambia il valore della _head della coda senza modificare l'anzianità dell'elemento. Nel caso la coda non abbia elementi viene generata una eccezione custom *empty_queue_exception*;
- **setTail**: cambia il valore della _tail della coda senza modificare l'anzianità dell'elemento. Nel caso la coda non abbia elementi viene generata una eccezione custom *empty_queue_exception*;
- **search:** permette di verificare l'esistenza di almeno un elemento avente un certo valore di tipo generico passato per parametro. Ritorna true se viene trovato, altrimenti false;
- **transformif:** funzione globale generica che permette la modifica degli elementi di una coda, con un operatore F, che soddisfano un certo predicato P.

E' stato inoltre fatto l'overload dell'operatore di ostream per permettere la stampa dei valori della coda dal più recentemente inserito al meno recente.

Iteratori

È stato implementato un iteratore di lettura e scrittura di tipo forward. La scelta implementativa del tipo di iteratore è stata fatta in considerazione della struttura dati proposta da progettare. Si suppone che l'accesso ad una coda parta dalla testa, quindi verrà usato il puntatore _head per iterare la struttura. Il puntatore _tail viene usato convenzionalmente per garantire che le operazioni di inserimento e cancellazione (come quelle di get e set dell'ultimo elemento) vengano eseguite in tempo costante. L'iterazione quindi deve essere unidirezionale.

Main

Nel file main.cpp vengono testati tutti i metodi della classe coda. È stato fatto uso sia di assert sia chiamate a ogni metodo implementato. E' stato deciso di fare la maggior parte di test, per semplicità, su una coda di dati di tipo int, ma sono anche state testate code di tipo std::string e code di tipo point per assicurarsi la regolarità della libreria. Sono stati creati vari metodi all'interno della classe per distribuire opportunamente il testing della libreria coda.h.