# Relazione progetto C++
# Giugno 2020

Nome: Davide

Cognome: Pietrasanta

Matricola: 844824

Mail: d.pietrasanta@campus.unimib.it

Tempistiche: Dal 14/05/2020 al 05/06/2020

----------< Introduzione >----------

Dopo un'accurata valutazione del problema, ho deciso di rappresentare la coda FIFO tramite una classe Queue che rappresenta un insieme di nodi. Per separare l'entità nodo dalla coda stessa ho creato una struttura dati per i nodi e una classe per le code, aventi entrambe tre valori.

----------< Tipi di dati >----------

Ho deciso, come detto poc'anzi, di separare nodi e code. Ho dunque creato una struttura dati **Node** per descrivere un nodo di una coda FIFO. Essa contiene tre attributi:

- **value** è il valore di tipo generico T scelto dall'utente in fase di inizializzazione
- **next** è il puntatore al prossimo nodo
- **prev** è il puntatore al nodo precedente

Node è stata resa privata e la sua esistenza viene dunque nascosta all'utente che andrà ad utilizzare la classe Queue.

La classe che implementa la coda FIFO, chiamata **Queue**, contiene tre attributi:

- **sz** è un unsigned int che contiene la dimensione della coda
- **root** è il puntatore al nodo più anziano (primo)
- **last** è il puntatore al nodo più giovane (ultimo)

Separando l'entità coda da quella nodo abbiamo un unico valore a rappresentare la grandezza della coda, così da avere un ottimizzazione della memoria. Inoltre in questo

modo si garantisce un accesso costante al nodo più anziano e a quello più giovane mantenendo tempi costanti anche per l'inserimento o la rimozione di nodi.

Tutti gli attributi di Queue sono privati in modo da non consentire la manipolazione non controllata della coda.

----------< Implementazione >----------

Come espressamente richiesto dalla consegna, non era possibile l'utilizzo di librerie esterne e strutture dati container della std library.

Ho dunque implementato i metodi essenziali per la classe Queue:

- **Costruttore di default**, che inizializza una coda vuota
- **Costruttore copia**, che inizializza una coda copia di un' altra
- **Costruttore secondario**, per inizializzare tramite un generico valore di tipo T, ovvero inizializza una coda di dimensione uno e con valore della radice pari al valore inserito.
- **Distruttore**, che elimina ogni allocazione effettuata in memoria durante il corso dello svolgimento del programma utilizzando una funzione di supporto *destroy()*, definita nella classe Node e privata.
- **operator =**, che permette assegnamento

----------< Metodi implementati >----------

Nella classe Queue:

- **size()** - E' stata richiesta una funzione per conoscere il numero totale di dati nella coda, ho dunque implementato la funzione size() che ritorna il valore dell'attributo sz presente nell'oggetto. Alla creazione viene inizializzato e ad ogni push() o pop() il valore viene aggiornato. Nel caso di assegnamento esso viene aggiornato. Ho deciso di creare un attributo apposito per la dimensione della coda in modo da velocizzare la funzione rendendola di complessità temporale costante.
- ***Getter & Setter*** – Ho implementato metodi getter per i valori della root, ovvero **get_older()**, e della last, ovvero **get_younger()**. Inoltre ho implementato metodi setter per i valori della root, ovvero **set_older()**, e della last, ovvero **set_younger()**. Tutti questi metodi possono genera un'eccezione **my_uninitialized_value()** in caso si cerchi di accedere a valori non ancora inizializzati e hanno complessità temporale costante.
- **push(const T &)** - Permette l'inserimento di un valore di tipo T sul fondo della coda. L'inserimento avviene in tempo costante grazie all'uso del puntatore last.

- **push(const const_iterator& i, const const_iterator& ie) –** Permette l'Inserimento di un insieme di elementi presi da una sequenza identificata da due iteratori. L'ordine di inserimento determina l'anzianità dell'elemento. Potrebbero esser generate le eccezioni **out_of_range_iterator()** e **my_uninitialized_value()** in caso gli iteratori puntino ad elementi non inizializzati oppure l'iteratore *i* sia successivo all'iteratore *ie* . Ha tempo di esecuzione lineare rispetto agli elementi inseriti.
- **push(const terator& i, const terator& ie)** – Permette l'Inserimento di un insieme di elementi presi da una sequenza identificata da due iteratori. L'ordine di inserimento determina l'anzianità dell'elemento. Potrebbero esser generate le eccezioni **out_of_range_iterator()** e **my_uninitialized_value()** in caso gli iteratori puntino ad elementi non inizializzati oppure l'iteratore *i* sia successivo all'iteratore *ie* nella coda presa. Ha tempo di esecuzione lineare rispetto agli elementi inseriti.
- **pop()** - Permette la rimozione dell'elemento più anziano della coda. Ciò ha tempo di esecuzione costante.
- ***Ricerca –*** Deve essere possibile chiedere alla classe se contiene almeno un elemento di un certo valore dato. Per questo fine è stata implementata la funzione **contain(const T& value)**. Essa ritorna true se la coda contiene almeno un elemento di un certo valore dato (value), altrimenti false. Ha tempo di esecuzione lineare rispetto alla lunghezza della coda.
- **Stampa –** E' possibile visualizzare la coda tramite la funzione **show()** che utilizza la funzione *show()* della struttura *Node,* che opera in modo ricorsivo. Se la coda è vuota viene segnalato e la stampa avviene tramite std::cout. Per funzionare la classe T deve dunque possedere **operator<<** .

Funzione globale:

- **trasformif(Queue<T> &q, P pred, F f)** - Prende in ingresso una coda con valori di tipo T, un predicato funtore di tipo P e un operatore funtore di tipo F. Modifica con l'operatore f gli elementi della coda che soddisfano il predicato p.

----------< Iteratori >----------

La specifica di progetto richiedeva l'implementazione di iteratori lettura E scrittura. Ho dunque implementato **const_iterator** e **iterator** di tipo forward con ordine di visualizzazione in accordo con la politica FIFO. I const_iterator permettono la sola lettura mentre gli iterator permettono sia la lettura che la scrittura. Per scorrere la coda si utilizzano i puntatori next presenti in Node. La push() opera sui next collegando l'ultimo next al nuovo elemento inserito.

----------< Main >----------

Nel file main.cpp vengono testati tutti i metodi pubblici resi disponibili dalla classe Queue. Vengono testate diverse modalità di utilizzo dei metodi e tramite assert viene verificata la

correttezza dei risultati. Come richiesto non è quindi necessario nessun input da parte dell'utente.

Vengono testati gli iteratori e la loro eccezione custom.

Vengono inoltre testate tutte le eccezioni custom. Ho ritenuto necessario crearne tre:

- **my_uninitialized_value** serve per controllare l'accesso a valori non ancora inizializzati.
- **out_of_range_iterator** serve per controllare l'accesso ad indici dell'iteratore maggiori di end().
- **bad_pop_exception** serve per controllare la rimozione da una coda vuota.

Ho provveduto anche a testare inizializzazioni della classe passando tipi diversi. Ho effettuato test con code di tipo int, string e una classe custom (Classe_custom).