Relazione progetto Multiset ordinato

Nome: Andrea
Cognome: Lamparella
Matricola: 829602
email: a.lamparella@campus.unimib.it

Descrizione del progetto

Il progetto consiste nello sviluppare una classe che implementa un multiset di elementi generici ordinati.

Un multiset prevede che più istanze di uno stesso elemento possono occorrere. Essendo ordinato l'inserimento di nuovi elementi viene fatto secondo una definita relazione d'ordine.

Scelte implementative

Vista la necessità di un utilizzo minimale della memoria, la struttura dati su cui si basa il multiset è costituito da un array dinamico di coppie <valore, occorrenze>. In questo modo l'aggiunta e/o rimozione di un elemento già presente nel multiset comporta solo l'aggiornamente del contatore delle occorrenze del relativo elemento.

Tuttavia sono comunque presenti delle situazioni in cui i tempi di esecuzioni sono ridotti a causa della necessità di un minimo uso di memoria. Per riportare un esempio, questo si verifica in caso di frequenti inserimenti di dati non già presenti nel multiset. Infatti, l'aggiunta di un nuovo elemento comporta l'allocazione di una nuova area di memoria contigua nello heap che permetta di ospitare un elemento in più rispetto alla precedente area di memoria, copiare i dati precedenti (quindi un tempo lineare rispetto ad alla dimensione dell'array) aggiungendo nella posizione corretta il nuovo elemento. In caso di informazioni supplementari sullo scenario di utilizzo della struttura dati sarebbe stato possibile determinare un fattore di crescita per ottimizzare il rapporto dimensione/spazio allocato.

Parametri template

- T : neccassario per permettere alla struttura dati di essere *generica*
- Comp : permette di specificare la relazione d'ordine secondo cui gli elementi devono essere ordinati
- Eq : permette di specificare la relazione di equivalenza con cui gli elementi vengono considerati uguali

Metodi implementati

La classe è dotata di metodi che sono stati sviluppati per mantenere la massima coerenza durante il ciclo di vita di ogni sua istanza. Le più importanti sono qui di seguito riportate.

`void insert(const element&)`

Inserisce un elemento all'interno del multiset. Se questo è già presente ci si limita ad aumentare il relativo contatore di occorrenze. Altrimenti si procede allocando spazio in memoria per ospitare il nuovo elemento. Dopo aver determinato la posizione in cui il nuovo elemento deve essere inserito vengono prima aggiunti gli elementi che precedono il nuovo elemento, secondo la relazione d'ordine specificata, poi questo e infine gli elementi che lo susseguono.

`void erase(const element&)`

Cancella un elemento all'interno del multiset. Se questo non è effettivamente presente nella struttura dati allora viene lanciata una eccezione `element_not_found`. Se, invece, l'elemento è presente e le sue occorrenze sono superiori (>) a una, si procede a ridurre il numero di occorrenze. In caso contrario viene allocato spazio in memoria, tenendo conto delle ora ridotte necessità di spazio, si individua la posizione in cui è presente l'elemento da eliminare e si procede a ricopiare gli elementi che lo precedono e che lo susseguono, secondo la relazione d'ordine specificata.

`size_type binsearch(const element&) const`

Per l'aggiunta e la rimozione di elementi, ma non solo, è necessario individuare prima il relativo indice all'interno dell'array dinamico. Per fare ciò si effettua una ricerca dicotimica che ritorna:

- se il valore è presente nel multiset, il suo indice
- se non è presente ritorna l'indice in cui si dovrebbe inserire l'elemento per mantenere l'ordine nella struttura dati
- esattamente la dimensione dell'array nel caso in cui il valore sia il più grande nel multiset

A causa della caratteristica che si evidenzia in quest'ultimo punto si effettuano diversi controlli ogni qual volta si usa la funzione `binsearch()` per accertarsi di non star accedendo ad aree di memoria al di fuori dell'array. Questa necessità di controlli permette di ridurre notevolmente i tempi di ricerca nel caso si fosse effettuata una più semplice ricerca lineare.

`size_type size() const`

Restistuisce il numero totale di elementi presenti nel multiset. È presente un controllo di "overflow" nel caso in cui la somma delle frequenze di tutti i dati sia superiore a quella rappresentabile da un unsigned int. Il "controllo di overflow" non è da interpetare in modo pedante poiché lo standard C++ prevede che le operazioni con i tipi unsigned siano pensate come modulo 2^n, dove n è il numero di bits per rappresentare quel particolare intero. Inoltre non è considerato *undefined behaviour* l'operazione UINT_MAX + 1, il quale risulta 0 (*wraps around*), cosa che non è garantita

per dati con segno per quali quindi è considerato *undefined behaviour* l'operazione INT_MAX + 1. Il controllo è stato inserito per voler comunque intercettari eventuali riavvolgimenti. Per questioni di coerenza con il corso si è preferito continuare ad usare il tipo `unsigned int` con la semantica di tipo di dato per l'indicizzazione, ma si è voluto comunque prevedere la possibilità in cui ci sia un overflow nel caso si dovesse cambiare tipo di dato.

```
template<typename Comp2> bool operator==(const multiset<T, Comp2, Eq> &other) const
```

Permette di controllare se due multiset sono uguali a meno della relazione d'ordine con cui questi sono istanziati. Pertanto due multiset che figurano gli stessi elementi, quindi stesse coppie <valore, occorrenze>, sono considerati uguali anche se sono ordinati in modo differente. Tuttavia, per essere confrontabili due multiset devono godere della stessa relazione di uguaglianza.

Iteratore

È stato scelto di implementare un bidirectional iterator per riflettere la navigabilità sequenziale che caratterizza un multiinsieme, che non sarebbe stata colta da un random iterator, su entrambe le direzioni visto che la struttura dati è ordinata ed è sempre possibile stabilire quale elemento sussegue o precede un'altro.

Eccezioni

- std::overfow : nella funzione size()
- element_not_found : eccezione custom lanciata ogni qual volta si cerca di rimuovere un elemento non presente nel multiset

Test implementati

Al fine di verificare la correttezza del programma sono stati scritti diversi test sia su tipi base che su tipi custom. Il meccanismo delle asserzioni è stato largamente usato per scrivere i test.

I test eseguiti consistono nel verificare il corretto stato dei dati membro della classe in seguito all'istanziazione con i costruttori disponibili (default, copia e con iteratori) e all'operatore di assegnamento. Le funzioni insert() e erase() vengono usate per testare i comportamenti nei casi di inserimenti/rimozioni in testa e in coda e le funzioni size(), contains(), occurs(), richieste dal progetto, vengon impiegate per verificare il corretto comportamento di queste operazioni. Ciò è ripetuto su dati const per controllare che i dati non siano effettivamente modificabili (attraverso insert() e remove()) e che sia valida la const correctness.

Sono stati scritti alcuni semplici funtori per i diversi tipi di dati su cui verranno effettuati i test, incapsulate in tre struct da cui il compilatore deddurrà automaticamente la funzione con la signature più adatta al contesto.

I dati custom realizzati sono `item`, cioè un ipotetico articolo di un inventario di magazzino, `str_pair`, una generica coppia di stringhe e `point2D`, la descrizione di un punto a coordinate intere nello spazio 2D. Ognuno di questi dati presenta delle funzioni per essere usato correttamente dalla classe, default constructor e operator<<, e dai funtori, cioè i vari operatori di confronto presenti.

Numerose volte si è usata la libreria standard per cercare di snellire il codice e renderlo più leggibile, oltre a essere un ulteriore test di come la classe multiset sia compatibile con la libreria standard. Si citano di seguito alcuni casi:

- std::vector : è stato largamente usato come contenitore di dati da usare per popolare i vari multiset
- std::set : usato principalmente per ottenere una struttura dati contenente le copie uniche di un vector, in modo da rendere più semplici le iterazioni sugli elementi dei multiset
- std::min e std::max : sono stati usati per ottenere rapidamente alcuni dei dati per effettuare i test con la funzione erase()
- std::fabs e std::numeric_limits : per eseguire in modo corretto la differenza fra dati di tipo double

Per testare l'eccezione custom prodotta si effettuano dei tentativi di rimozione di elementi non presenti andando però a intercettare l'eccezione e continuando con l'esecuzione del programma.

Compilazione

Il codice è stato testato su windows e linux con i seguenti compilatori: windows 10: g++ 12.2.0 linux : g++ 11.1.0