Relazione progetto C++

A.A 2019/2020

Sessione di Aprile 2020

BINARY SEARCH TREE

Nome: Matteo

Cognome: Fornara

Matricola: 829545

Mail: m.fornara@campus.unimib.it

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

La traccia del progetto richiede la realizzazione di una classe generica che implementa un albero binario di ricerca. L'albero è formato da un insieme di dati di tipo T (ossia il tipo generico con il quale si crea l'albero), i quali sono contenuti in nodi connessi tra loro in una struttura gerarchica padre-figlio.

Fondamentale che l'albero non accetti dati duplicati, e che sia possibile per l'utente scegliere la strategia da usare per confrontare l'albero.

Inoltre l'albero deve implementare qualche altra funzione che andrò a spiegare e a commentare come io l'abbia gestita, nel corso di questa relazione.

<Tipi di dato utilizzati>----------

Ho deciso di definire due nuovi tipi di dati tramite typedef per rendere più leggibile il codice, questi nuovi tipi sono:

- **size_type:** E' un unsigned int, che ho utilizzato all'interno del codice per la dimensione dell'albero (_size).
- **value_type:** E' il tipo generico T scelto dall'utente in fase di inizializzazione dell'albero, e quindi il tipo di dato dei valori all'interno dell'albero.

L'altro tipo di dato all'interno del mio codice è la struttura Node, la quale è la struttura interna di supporto, contenitore dei valori inseriti nell'albero, i quali dati membro sono:

- **value:** Il dato inserito nel nodo, e quindi all'interno dell'albero , di tipo value_type, quindi del tipo corrispondente al tipo con cui è stato costruito l'albero.
- **left:** Puntatore al nodo figlio di sinistro, anche questo di tipo Node.
- **right:** Puntatore al nodo figlio di destra, anche questo di tipo Node.
- **parent:** Puntatore al nodo parent, anch'esso di tipo Node.

La struttura Node è il modo con il quale ho gestito i dati inseriti nell'albero, creando grazie ad essa una gerarchia padre-figlio richiesta nella traccia.

La struttura l'ho definita privata all'interno della classe, così da evitare la creazione di nodi indipendenti al di fuori di un qualsiasi albero. Saranno i metodi della classe a gestire i nodi, a crearli, a distruggerli, e a modificarli.

Come funzioni membro, la struttura Node contiene:

1. Costruttore di default, il quale inizializza il nodo ponendo tutti i campi della struct a nullptr.
2. Costruttore secondario, il quale inizializza il nodo parzialmente con un valore (value) passato come parametro.

3. Costruttore secondario, il quale inizializza il nodo completamente, con il valore che contiene, e i rispettivi nodo figlio sinistro, nodo figlio destro e nodo parent.
4. Costruttore secondario, il quale inizializza il nodo parzialmente, con figlio sinistro, figlio destro e valore.
5. Distruttore, il quale pone a nullptr i tre puntatori (left, right, parent).

Tutti queste funzioni membro, essendo la struttura privata, sono utilizzabili solamente all'interno della classe.

Sugli altri metodi fondamentali ho ritenuto inutile farci l'overriding, e ho lasciato quelli di default.

<Commento implementazione>----------

Nodo (non dell'albero) fondamentale di questo progetto è stato la decisione su come gestire la scelta della strategia di confronto dei dati all'interno dell'albero da parte dell'utente. La strada che ho preso per rendere possibile questo è stata quella di aggiungere alla classe due parametri template:

- **C** riferito al funtore di comparazione (<) di due dati.
- **E** riferito al funtore di uguaglianza (==) di due dati.

Questi parametri template permettono all'utente, alla creazione di un qualsiasi albero di scegliere le strategie di confronto che preferisce, passando semplicemente dei funtori alla creazione, esempio:

Bst<int, funtore_confronto_interi, funtore_uguaglianza_interi> albero;

Dove **int** è il tipo template T, **funtore_confronto_interi** è il tipo template C, e **funtore_uguaglianza_interi** è il tipo template E.

Definendo vari funtori, coerenti con il tipo generico T utilizzato per la creazione dell'albero, si possono definire diverse strategie di confronto dati e ciò andrà a modificare il modo in cui l'albero posizionerà i dati, e quindi i rispettivi nodi all'interno dell'albero.

La classe Bst contiene 4 dati membro privati:

1. **_head:** Il puntatore al nodo radice dell'albero, di tipo Node.
2. **_size:** Numero di dati, e quindi di nodi all'interno dell'albero.
3. **conf:** Oggetto funtore per il confronto.
4. **eql:** Oggetto funtore per l'uguaglianza.

Altra scelta implementativa che ho fatto è stata quella di lasciare la possibilità all'utente di creare alberi tramite tre costruttori:

- **Costruttore di default:** Il quale crea un albero vuoto, ponendo _head a nullptr e _size a zero, non contenendo nessun nodo.
- **Costruttore secondario:** Il quale inizializza l'albero inserendo tramite la funzione add() tutti i dati presenti in un array passato come parametro.
- **Costruttore di copia:** Il quale permette di inizializzare un nuovo albero con i valori presi da un altro, in poche parole facendoci una copia deep. Esso utilizza una funzione helper privata (helper_copy_cstr), la quale, passatogli come parametro la radice (_head) dell'albero da copiare, naviga lo stesso, ricorsivamente, in maniera pre-order, e aggiunge con il metodo add() tutti i dati. Il fatto di navigare l'albero in pre-order è necessario affinchè l'albero risultante abbia la stessa struttura dell'albero copiato.

Come si sarà notato da come ho implementato i vari costruttori, una scelta implementativa è stata quella di far passare dalla funzione add() ogni inserimento di dati all'interno di un albero. Questo perché la funzione gestisce adeguatamente l'inserimento di un dato passatogli come parametro, ponendolo in un nodo, e posizionandolo all'interno dell'albero seguendo le strategie di confronto scelte dall'utente e lanciando le adeguate eccezioni in caso di errori. La _size di un albero, ogni volta che viene inserito un elemento, viene incrementata così da permettere all'utente di conoscere il numero di dati all'interno di un albero, semplicemente richiamando il metodo get_size().

Andando più nel dettaglio la funzione **add()** partendo dalla radice dell'albero su cui è chiamato, tramite i funtori, confronta il valore di un nodo con il valore del dato passatogli come parametro, e sempre dipendentemente dalla strategia di confronto scelta dall'utente tramite i funtori, sceglie se scendere l'albero a sinistra o a destra. Continua a scendere in questa maniera l'albero, fino a quando il nodo successivo a cui doveva passare è vuoto (nullptr). A questo punto crea un nuovo nodo con il valore da inserire e sceglie se porlo figlio sinistro o figlio destro dell'ultimo nodo che ha incontrato.

E' possibile che la funzione add() possa lanciare due eccezioni, un'eccezione di allocazione di memoria, oppure l'eccezione custom creata da me (duplicateDataException), la quale viene lanciata se la funzione add dovesse trovare qualche nodo all'interno dell'albero che abbia il valore (value) uguale a quello passato come parametro. Ciò impedisce che l'albero accetti dati duplicati, funzione fondamentale richiesta dalla specifica del progetto.

Oltre ad utilizzare i costruttori l'utente potrebbe creare un albero, inizializzandolo con un costruttore a scelta e poi utilizzando il metodo fondamentale di assegnamento (operator=) per renderlo uguale ad un altro albero. Anche l'operatore di assegnamento si fa aiutare dal metodo helper_copy_cstr, per creare un albero di copia, e poi renderlo quello di riferimento tramite la funzione swap() della libreria standard.

Infine come scelta implementativa ho deciso, nel distruttore, di chiamare il metodo **clear()**, il quale viene richiamato anche in caso di eccezione di allocazione di memoria, e che si appoggia ad un metodo helper privato (helper_clear), che, passatogli come parametro il nodo radice dell'albero da eliminare, naviga lo stesso, ricorsivamente, in maniera post-order, cancellando dalla memoria tutti i nodi dell'albero. La navigazione dell'albero in post-order è necessaria affinchè vengano cancellati per primi i nodi esterni evitando di lasciare nodi, o addirittura sottoalberi orfani.

<Altri metodi richiesti>----------

I. **get_size:** Ritorna semplicemente il dato membro _size, il quale segnala quanti dati sono presenti all'interno di un albero.

II. **exist:** Con l'aiuto degli iteratori (ne parlerò nel prossimo paragrafo), naviga l'intero albero in cerca del valore passato come parametro, nel caso lo trovasse ritornerebbe un booleano true, in caso contrario un booleano false.

III. **printTree:** Stampa l'albero la cui radice è stata passata come parametro, grazie all'aiuto di un metodo helper (helper_print), il quale, ricorsivamente, processando prima i rami destri dell'albero e poi quelli sinistri, stampa orizzontalmente il contenuto dell'albero, seguendo una struttura simil albero binario.

IV. **operator<<:** Funzione globale la quale, passatole come parametri il riferimento all'oggetto ostream di output (os) e un albero, stampa il contenuto di quest'ultimo utilizzando il metodo membro printTree() passandogli il nodo radice dell'albero.

V. **subtree:** Utilizzando gli iteratori (ne parlerò nel prossimo paragrafo), naviga l'intero albero in cerca di un nodo con valore uguale a quello passatogli come parametro, se lo trova ritornerà il sottoalbero con, come radice, il nodo trovato. Anche questo metodo si appoggia al metodo helper_copy_cstr, passandogli il nodo trovato.

VI. **printIF:** Funzione globale che dato un albero binario di tipo T, e un predicato P (il quale sarà un funtore), stampa a schermo tutti i valori contenuti nell'albero che soddisfano il predicato passato come parametro. Il metodo svolge la sua funzione aiutato dagli iteratori; naviga grazie a questi l'intero albero, e stampa i valori dei nodi che soddisfano il predicato.

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

La specifica di progetto richiedeva l'implementazione di un iteratore di tipo forward a sola lettura (quindi const_iterator). Ho deciso di implementare un iteratore che navighi l'albero in-order, prendendo cioè i nodi in ordine, dal minore, al maggiore.

Riesco a fare ciò grazie all'algoritmo che sta dietro ai metodi di pre-incremento e post-incremento.

Se il nodo corrente ha un figlio destro, passa a questo e scende fino al nodo minore tra i figli sinistri, ed esso sarà il nodo successivo.

Se invece il nodo corrente (chiamiamolo x) non ha figli destri, ci si muove su per l'albero, cercando il primo nodo che abbia x nel suo ramo sinistro, questo sarà il successore della sequenza. Se questo nodo invece non si trova, uscendo dall'albero, x è l'ultimo nodo della sequenza.

Il metodo begin() appoggiandosi al metodo findMin(), il quale trova il valore "più a sinistra" dell'albero ossia quello più piccolo (dipendentemente dalla strategia di confronto), ritorna l'iteratore su quello, il nodo più a sinistra, il quale sarà il nodo iniziale della sequenza dati.

Il metodo end() ritorna invece l'iteratore all'ultimo nodo della sequenza, il nodo vuoto (nullptr).

<Tests>----------

Nel file main.cpp vengono testati tutti i metodi pubblici resi disponibili dalla classe Bst. Vengono testate diverse modalità di utilizzo dei metodi e tramite assert viene verificata la correttezza dei risultati, non è quindi richiesto nessun input da parte dell'utente.

Nel file creo anche tutti i vari funtori utilizzabili come strategie di confronto dati, per tipi di dato String, int, char e Book, quest'ultimo un tipo custom (una struct) che rappresenta un libro, con il numero di pagine e il suo codice (definendo per questo anche l'operatore<<).

Oltre a questi funtori ne ho definiti altri due da utilizzare come predicato nel metodo printIF:

- **is_even:** Il quale controlla se il valore passatogli come parametro è pari.
- **big_book:** Il quale controlla la dimensione di un libro, se le pagine del libro passato sono più di 500, ritorna true, altrimenti false.

<Informazioni aggiuntive>----------

L'operatore di assegnamento, il metodo helper_copy_cstr, il metodo subtree e il copy constructor non lanciano la duplicateDataException poiché avendo a che fare con alberi già creati, essi non possono contenere dati duplicati a priori.