Università degli studi di Milano Bicocca
Dipartimento di Informatica, Sistemistica e Comunicazione
Progetto del corso di Programmazione C++

Relazione Progetto C++

Gianlorenzo Occhipinti
g.occhipinti3@campus.unimib.it
829524

Milano, Aprile 2020

Indice

1. Progetto C++

Dopo un'analisi delle specifiche del progetto per la realizzazione di un BST che lavora con tipi generici, ho scelto di implementare i nodi in una "struttura dati" (struct), che contiene i riferimenti al nodo padre/destro/sinistro e il valore del nodo; la classe BST invece rappresenta l'interfaccia pubblica del progetto; essa si occupa della gestione trasparente ed esclusiva dei nodi.

1.1 Tipi di dati

La classe BST si serve di 3 tipi di dati:

1. Il tipo **T** di dato usato per caratterizzare i valori dei nodi, passato come parametro del template durante la dichiarazione e istanziazione dell'albero.
2. L'operatore di confronto **C**, implementato tramite funtore passato come parametro del template durante la dichiarazione e istanziazione dell'albero, con lo scopo di dettare un ordinamento tra i nodi.
3. IL **nodo**, una sottostruttura fondamentale che contiene le informazioni dell'albero

1.1.1 Nodo

I nodi sono la componente chiave di un albero; nella mia implementazione ho deciso di dichiarare i nodi come struct e non come class perché viste le caratteristiche di un nodo (membri tutti pubblici) ho ritenuto fosse la scelta più sensata.

Per poter avere una agevole navigazione dell'albero ho deciso di salvare in un nodo:

- Puntatore al nodo padre, per poter salire a ritroso dall'albero e non dover partire sempre dalla radice per raggiungere un altro nodo.
- Puntatore al nodo destro.
- Puntatore al nodo sinistro.
- Valore del nodo.

Tutti i nodi risiedono nello heap e la loro costruzione (e cancellazione), seguendo il pattern R.A.I.I, spetta esclusivamente alla classe BST (che quindi ha la responsabilità della loro corretta gestione).

La *struct* del nodo è dichiarata privata all'interno della classe BST così da non permettere ad "estranei" (quindi classi non dichiarate *friend*) la manipolazione di essi.

1.1.2 Operatore di confronto

Un punto dei requisiti del progetto è di dare all'utente la possibilità di adottare una proria strategia di confronto dei dati. Questo è possibile farlo definendo un **funtore** da passare come parametro template alla classe BST.

Il funtore però non deve tornare un booleano come intuitivamente si può pensare ma uno **short**. La scelta è dettata dal fatto che voglio dare la possibilità agli utilizzatori della mia classe di poter definire una propria strategia anche per l'uguaglianza fra i 2 tipi di dati, senza doversi per forza appoggiare all'utilizzo degli operatori predefiniti. In questo modo si possono scrivere confronti più efficaci (ad esempio classi con un attributo simile ad una chiave primaria devono confrontare solo quel campo e non tutti) anche qualora non fosse possibile (oppure voluto) l'override dell'*operatore=*.

1.2 Albero binario di ricerca

Nella classe BST i metodi che sono dichiarati const garantiscono il rispetto della *constness* in quanto non alterano lo stato della classe. Nella firma dei metodi si nota come il tipo T sia passato per reference; questa scelta è stata presa per prevenire copie di dati onerose qualora il tipo T rappresenti una classe che occupi molto spazio in memoria. Tutti i metodi della classe sono autoesplicativi; onde evitare di rendere troppo tediosa la lettura di questa relazione, mi soffermerò solo sui metodi essenziali.

1.2.1 Metodi fondamentali

L'implementazione della classe BST, come richiesto nella consegna, è avvenuta mediante la dichiarazione di una classe **templata** con 2 parametri:

- il tipo **T** dei dati contenuti nell'albero.
- un funtore **C** per il confronto dei dati.

Per rendere comodo l'utilizzo di questa libreria ho ideato un operatore di confronto di default che, se non diversamente specificato in fase di istanziazione dell'albero, fa un semplice confronto tra i 2 operandi chiamando l'operatore relazionale $>$ (val1 $>$ val2); in questo modo è possibile dichiarare un albero comunicandogli solo il tipo di dato.

Successivamente ho pensato all'implementazione dei metodi fondamentali:

Il costruttore di default, non potendo ricevere parametri in ingresso si occupa di istanziare un albero, senza un nodo radice di dimensione 0.

A supporto del costruttore di default, per comodità, ho scritto anche un costruttore secondario che prende in input il dato che diventerà la radice dell'albero.

Il copy constructor ricorsivamente guarda, i valori da sinistra verso destra e li aggiunge a se stesso a mano a mano che scorre i nodi. Se dovessero essere lanciate eccezioni durante questa fase molto "delicata", prima di inoltrare l'eccezione al livello superiore, cancella tutti i nodi creati fin'ora così da evitare potenziali memory leak.

L'operatore di assegnamento sfrutta il copy constructor sopracitato per operare; una volta creato il clone scambia le variabili root e size così da appropriarsi dei valori del clone (quindi della variabile da copiare).

Il distruttore, infine, cancella ricorsivamente tutti i nodi aggiunti all'albero durante l'esecuzione del programma. Valgrind ha confermato la correttezza del suo operato non rilevando potenziali problemi per la memoria.

Per evitare di incombere in codesmell di codice duplicato ho deciso di scrivere dei metodi di supporto [1.2.3] poi chiamati da vari metodi all'interno della classe.

1.2.2 Metodi richiesti

Il primo metodo richiesto è quello per sapere le dimensioni dell'albero: questo metodo ha un'implementazione banale in quanto la dimensione dell'albero è un dato membro della classe (quindi non va calcolato ogni volta). Dietro la scelta di avere la dimensione come dato membro della classe si nasconde la necessità di dare alla classe BST la gestione esclusiva dei nodi, infatti la classe ad ogni inserimento di un dato mediante il metodo fornito, incrementa il valore della sua dimensione; se la classe non avesse il controllo esclusivo della gestione dei nodi, il suo contatore della dimensione diverrebbe inaffidabile.

Il controllo di esistenza di un elemento T è stato implentato nel metodo *contains*, il quale effettua una ricerca ricorsiva fra i vari nodi basandosi sul risultato fornito dall'operatore di confronto della classe (che ricordo può essere o di default oppure personalizzato). Si fermerà soltanto se troverà un nodo nullo (quindi ricerca con esito negativo) oppure se l'operatore di confronto [ref.1.1.2] restituisce il valore 0 (quindi ricerca con esito positivo).

L'iteratore di sola lettura di tipo forward è stato implementato come un tipo di dato interno alla classe. Per garantire la "sola lettura" chiesta nella cosnegna mi sono limitato a implementare solo un *const_ iterator* indiicando nei traits che puntatore e reference fossero di tipo costante così da garantirmi l'immutabilità dell'oggetto (se dovessero cambiare i valori dei nodi rischierebbero di compromettere la struttura dell'albero). L'iteratore è in grado di scorrere i dati in ordine crescente. Per implementare ciò mi sono limitato ad osservare i comportamenti da effettuare sull'albero per avere un ordine crescente andando da sinistra a destra e ho creato queste 4 regole:

1. Se salgo da destra continuo a salire.
2. Se salgo da sinistra mi fermo.
3. Mi muovo sempre verso destra.
4. Se sto scendendo e il nodo di sinistra è vuoto mi fermo.

Riassumendo: l'iteratore fa una visita in-order dell'albero.

La stampa del contenuto dell'albero si limita a scorrere l'albero con l'iteratore per stampare i valori dei vari nodi.

Per implementare il metodo *subtree* ho pensato di sfruttare la stessa funzione di supporto chiamata dal copy-constructor. Visto che entrambi i metodi devono tornare la copia di un albero a partire da un nodo (nel caso del copy constructor dal root, nel caso di *subtree* dal nodo del dato passato), ho pensato per prima cosa di costruire un albero vuoto per poi chiamare il metodo di supporto *recursive_ copy* passandogli in input il puntatore al nodo del dato in input (trovato con un'altra funzione di supporto *find_ node*). Se si passa in input un dato che non esiste l'albero tornato sarà vuoto.

Infine il metodo *printIF* il cui scopo è quello di stampare tutti i dati che rispettino un predicato **P** passato come parametro non fa altro che iterare tutti i nodi dell'albero per poi stampare solo quelli il cui predicato risulti verificato.

1.2.3 Metodi di supporto

Il primo metodo di supporto implementato è stato *sub_ tree* che, passato il puntatore ad un nodo di un altro bst, aggiunge a se stesso il valore di tutti i nodi appartenenti al sottoinsieme di nodi passati come paramentro.

Un altro metodo di supporto messo a disposizione per la classe bst è *find_ node*, che si limita a cercare iterativamente se esiste un nodo il cui valore sia uguale (quindi usando l'operatore di confronto istanziato) a quello del da passato come parametro della funzione.

L'aggiunta di un nodo all'albero è realizzata mediante il metodo *add_ node*, che garantisce il rispetto delle proprietà che caratterizzano un albero binario di ricerca. Se dovesse provare ad aggiungere un nodo duplicato (quindi un nodo il cui operatore di confronto torni 0) lancia un'eccezione [1.2.4].

Infine l'ultimo metodo di supporto messo a disposizione per i test è *height* che calcola ricorsivamente l'altezza dell'albero sommando i vari livelli e tornando sempre il più alto trovato.

1.2.4 Eccezioni custom

Per garantire all'utilizzatore della libreria un'esecuzione controllata di ciò che succede durante l'esecuzione della classe, se dovessero essere inseriti dati duplicati, non posso permettermi di ignorarli ma devo comunicare che c'è stato un tentativo di inserimento **illegale**.

L'eccezione *bst_duplicated_value_exception* viene lanciata in queste circostanze.

Non ho ritenuto necessario mettere il dato che ha generato l'eccezione come membro della classe perché questo è un dato di cui dispone già il programmatore nel momento in cui viene segnalata l'eccezione su quella linea di codice.

1.3 Main

Il main, contenuto nel file *main.cpp*, è strutturato in questo modo:

- Fase1: Test di tutti i metodi con della classe BST.
- Fase2: Test di un bst di interi con operatore di confronto di default [1.1].
- Fase3: Test di una bst con un tipo di dato custom chiamato "record" (una coppia stringa, double) con operatore di confronto custom.

Ciascuna fase prevede vari controlli, tutti stampati a video in modo da rendere molto chiaro che cosa si sta testando.

Un controllo per essere "Passato!" deve aver superato un asserzione che verifica se si è in presenza del risultato atteso.

Nella fase 2 di test viene usato l'albero in figura 1.1. Viene testato inizialmente il costruttore secondario, l'aggiunta di valori all'albero, viene controllato se è correttamente rilevato un inserimento duplicato, viene stampato a video il contenuto dell'albero per poi andare a testare gli iteratori e il metodo subtree (confrontando gli indirizzi puntati dai 2 iteratori fermi sul nodo 10, se sono l'indirizzo dei nodi puntati è diverso subtree torna una copia come è giusto che sia); infine viene testato il metodo printIF che stampa a video solo i nodi con valore pari.

Lo stesso set di test viene effettuato durante la Fase3 con la differenza che in questo caso il dato di un nodo è una *struct* con 2 dati: una *string* che sarà la chiave e un *double* che sarà il valore. L'operatore di confronto fa una sottrazione dei 2 dati ed effettua un troncamento ad intero; con ciò consegue che 7,9 e 8,4 sono due valori UGUALI ($7,9 - 8,4 = -0,5 \implies 0$).

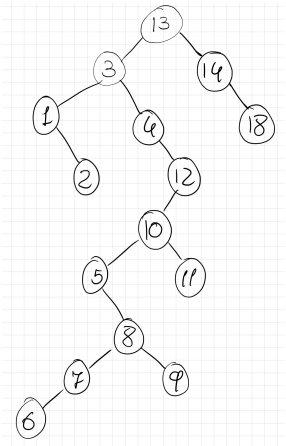

Figura 1.1: Albero del test nella fase 2