Binary Search Tree

Relazione del Progetto C++ - Aprile 2020

- Mattia Sgro
- 829474
- m.sgro2@campus.unimib.it

Introduzione

Un primo ragionamento sul problema proposto mi ha fatto subito pensare di implementare l'albero richiesto (un BST) tramite una struttura padre-figlio in cui ogni nodo potesse avere al più due figli (un *left* e un *right*) e fosse percorribile solamente dalla radice fino alle foglie.

Una successiva riflessione, seguita da varie scelte implementative, mi ha portato a definire, per ogni nodo, anche un puntatore al *parent* e al suo successore logico (*next*).

Il vantaggio dell'uso di puntatori *left* e *right*, oltre a rispecchiare a livello logico l'idea di un BST, consiste in un accesso in tempi minori alle celle della suddetta struttura dati (teoria di un albero binario di ricerca). I puntatori a *parent* e *next* si sono rivelati necessari per una soddisfacente implementazione dell'iteratore costante.

La struttura dati definitiva combina quindi un albero binario e una lista linkata.

Non è prevista nell'implementazione alcun metodo di bilanciamento dell'albero, essendo l'ordine di aggiunta degli elementi la peculiarità di ciascuna istanza del BST.

La Struttura dati

La mia implementazione del **binary search tree** presenta:

- Attributi `private` dell'albero
  - ***_root*** : un puntatore a nodo contenente la root dell'albero;
  - ***_size***: un membro che tiene traccia del numero di elementi dell'albero
  - ***compare_t*** : un funtore che permette la comparazione(<) di dati di tipo T
  - ***equal_t*** : un funtore che permette la comparazione per eguaglianza(==) di dati di tipo T
- una `struct` ***node*** : che forma i nodi dell'albero, ciascuno avente:
  - il valore da salvare (di tipo T) definito `const` perché immutato dopo la creazione del nodo
  - i puntatori a nodo *left*, *right*, *parent* e *next*

Per la *sottoclasse* `struct node` l'unico *costruttore* prevede di istanziare un nodo dato il valore in ingresso. Tale costruttore viene utilizzato solo dal metodo add() per aggiungere tale nodo al BST; il *distruttore* ovviamente setta a nullptr tutti i puntatori del nodo. L'esistenza di nodi vuoti in effetti non è necessaria.

Per il BST abbiamo quindi:

- un **costruttore di default** che permette di istanziare un albero vuoto (`_size=0`) e con puntatore alla radice vuoto (`_root=nullptr`), che verrà aggiornato durante la prima `add()` effettuata sull'albero

- un **costruttore secondario** che permette di istanziare un albero fornendo il valore del nodo radice; il costruttore è reso esplicito per evitare venga usato in cast di tipo non desiderati (essendo inoltre non richiesti i cast tra alberi di tipi diversi)
- l'overload del **costruttore di copia** che, attraverso il metodo `copy_helper` , esegue un copia fisica dei dati in un altro BST e permette la creazione di alberi a partire da altri già istanziati, dello stesso tipo
- l'overload dell'**operatore di assegnamento** che richiama il costruttore di copia per assegnare un albero ad un altro dello stesso tipo
- un **distruttore** che richiama una funzione di cancellazione dei nodi dell'albero, `trim_tree()` , la quale a sua volta procede in modo ricorsivo tramite la `trim(node *n)` a cancellare il nodo e tutti i suoi figli
- un **getter** per la size dell'albero
- le funzioni di **add**, **exist**, **print** e **subtree**

copy_helper()

A partire da un nodo in ingresso, questa funzione aggiunge tale nodo all'albero corrente e continua l'esecuzione sui nodi left e right.

Ho scelto di implementare in questo modo un copy_helper in quanto lo scopo primario era di usarlo partendo da un nodo di un albero esterno, per permetterne la copia in cascata di tutto l'albero. Questo metodo infatti viene utilizzato dentro la funzione subtree e nell'overload dell'operatore di copia del BST.

trim_tree()

Cancella l'albero chiamando con argomento _root la funzione ricorsiva trim(), che accetta in ingresso un nodo dal quale far partire una delete ricorsiva sui nodi left e right.

Viene lanciata nel distruttore del BST oppure, in caso di eccezioni, sull'albero corrente.

Funzioni

ADD

La funzione principale della classe, l'unica azione di modifica disponibile per l'utente su un oggetto di tipo binarytree(), è la funzione di aggiunta di un nodo `add(const valore)` tramite valore. L'inserimento pretende di confrontare tale valore con quelli già presenti nell'albero binario di ricerca, per aggiungerlo nella posizione corretta (corretta rispetto ad un BST, ovvero, preso un nodo, presenterà nel sottoalbero di sinistra solo nodi di valore minore e a destra solo nodi di valore maggiore). La comparazione dei valori dei nodi è resa possibile grazie ai funtori di comparazione ed eguaglianza definiti nel main dall'utente; è quindi egli stesso a scegliere come confrontare il tipo di dati inseriti. Come anticipato nel costruttore del BST, la add su un albero vuoto, aggiunge il nodo come root. In caso l'albero non fosse vuoto parte con il confronto del nodo dalla root e in cascata ai nodi figli:

- se il valore in aggiunta risulta minore del nodo corrente, passo al nodo left rispetto al nodo corrente: se risulta essere un nodo(puntatore diverso da `nullptr`) ripeto il confronto; se risulta essere un `nullptr` aggiungo in questa posizione un nuovo nodo con il valore di aggiunta
- se il valore in aggiunta risulta maggiore del nodo corrente, passo al nodo right del nodo corrente: come sopra aggiungo in caso di `nullptr` o ripeto la procedura
- se il valore in aggiunta risulta già presente nella posizione corrente, viene lanciata una `ExistingNodeException` e una `trim_tree()` per evitare leak, non essendo infatti possibile

avere nodi dello stesso valore nell'albero.
- nel caso non si riesca ad effettuare tale confronto, ovvero non risulta nell'albero un nodo uguale, maggiore o minore (il maggiore viene gestito come [non maggiore e non uguale]), viene lanciata `UnmatchableException` oltre alla `trim_tree()`. Un problema di questo tipo indica una non completezza nel coprire un confronto dei due funtori definiti nel main per il tipo corrente. Si è rivelato necessario inserire questa eccezione dopo prove effettuate nel main sul tipo custom Point2d.

Durante l'operazione di aggiunta viene tenuto in memoria tramite un booleano `(if true->right; else ->left)` l'ultima direzione presa dal nodo durante l'aggiunta. In questo modo è possibile assegnare correttamente il puntatore *parent* del nuovo nodo rispetto all'ultimo nodo visitato e i puntatori *left* o *right* del parent in base all'ultima direzione presa.

Alla fine di tutta la procedura viene lanciato il metodo privato `nexter()` e viene aggiornata la `_size`.

nexter()

Il metodo `nexter()` permette, tramite una chiamata al metodo `next()`, di assegnare ad ogni nodo, partendo dal minimo valore dell'albero, il puntatore al nodo logicamente successivo. Continua l'esecuzione fino a quando `next()` non ritorna un nullptr.

next()

Il metodo `next()` procede con il trovare un successore al nodo in input:

- se il nodo non ha un right, il successore si puo trovare nel primo parent con valore maggiore rispetto al nodo corrente.
    - se tale parent non viene trovato, il nodo corrente è automaticamente il max, la funzione restituisce quindi un nullptr
- al contrario se dovesse avere un right, mi sposto su quel nodo e attraverso un while vado nel nodo piu a sinistra; se non ci dovesse essere un nodo a sinitra, il successore è il nodo corrente a destra del nodo di input

min()

cerca il minimo dell'albero: partendo dalla root è il nodo più a sinistra; se la root non dovesse avere nodi a sinistra, è la root stessa; se l'albero fosse vuoto, restituisce `nullptr`

EXIST

Controlla con le stesse modalità di ricerca spiegate nel metodo add, se un elemento esiste o meno nell'albero. Non ho voluto implementare una funzione di ricerca da poter riutilizzare nella `add` e `nell'exist` per definire più specificatamente gli errori lanciati da entrambe.

Lancia solo l'eccezione `UnmatchableException` in caso di fallita comparazione.

Restituisce `true` in caso si sia trovato l'elemento nell'albero, `false` in caso contrario.

PRINT

La `print` viene effettuata in 3 modi:

```
- chiamata alla funzione `ptree()`
- chiamata alla funzione `verbose_ptree()`
- overload dell'operatore <<
```

ho pensato di poter far scegliere all'utente di avere una print più snella (`ptree`) rispetto ad una più dettagliata `verbose_ptree()` dell'albero.

ptree()

Effettua una chiamata al metodo ptree_helper() per la stampa ricorsiva dell'albero

L'output si presenta come una visita IN ORDER dell'albero e viene correttamente parentesizzato. La lettura viene facilitata nel mostrare il valore della root dell'albero.

verbose_ptree()

Effettua una chiamata al metodo verbose_ptree_helper() per la stampa ricorsiva.

L'output si presenta come una visita in PREORDER dell'albero, stampando un nodo per riga con i riferimenti ai puntatori `left right parent` se diversi da `nullptr`

operator<<

Banalmente cicla l'iteratore e stampa gli elementi in sequenza

CONST_ITERATOR

Per l'iteratore costante ho deciso di ciclare i valori dal minore al maggiore (sempre secondo l'operatore di comparazione definito dall'utente per il tipo di dati corrente), nonostante la specifica di non considerare tale dettaglio. Per raggiungere lo scopo ho dovuto ampliare la `struct node` come specificato nell'introduzione con il puntatore al nodo next. Il metodo next(), attraverso nexter() chiamato alla fine di ogni add(), permette quindi la riuscita di uno scorrimento ordinato settando per ogni nodo il puntatore al suo successore logico.

Per il resto la classe è una normale implementazione del solo iteratore costante, di tipo forward, con annessa ridefinizione degli operatori di copia, assegnamento, incremento, begin(), end(), ecc, rispettando la const correttezza che tale iteratore richiede.

Il nodo membro dell'iteratore è ovviamente anch'esso const.

SUBTREE

Sfrutta il metodo di scorrimento di `exist` e `add`, oltre alla funzione di appoggio `copy_helper`. In questo metodo effettuo una ricerca del valore in input nell'albero corrente e, a partire da tale nodo, lancio l'helper di copia. Questo metodo quindi ritorna un nuovo albero subtree, secondo la root desiderata e mantiene corretti tutti i puntatori (essendo infatti il nuovo albero generato da varie chiamate di add nel metodo copy_helper). Viene gestita in caso un'eccezione lanciata da copy_helper e cancellato quindi il nuovo albero tmp creato dalla funzione subito dopo la chiamata.

PRINT_IF

Implementata seguendo la traccia come una funzione globale templata che a partire da un binary tree templato e un funtore " predicato " permette, attraverso l'uso del const_iterator definito, di stampare i soli valori che soddisfano tale predicato, attraverso un controllo if durante lo scorrimento dei dati

Test (main)

I test nel main sono divisi in 4 funzioni principali, una per ogni tipo di albero definito nelle typedef della classe. Ho testato il BST su interi, double, stringhe e una classe custom di punti 2d per cui è stato necessario definire l'operatore di stampa << .

Sono presenti inoltre i funtori per ogni tipo di BST che definiscono la compare e l'equal.

Sono presenti due funtori per la print if tra interi , even e odd che stampano rispettivamente interi pari e interi dispari.