6/10/2021

Giacomo Ettore Rocco, [SM3500479], s277891@ds.units.it

Introduzione:

Il progetto consisteva nell' implementare una classe template *binary search tree* in c++, avendo cura di inserire internamente un iteratore e senza la creazione di memory leak, ovvero dei "buchi" di memoria non più utilizzati e non più liberabili.

Un *binary search tree* è un albero binario di ricerca, dove ogni nodo ha un nodo figlio sinistro e un nodo figlio destro, rispettivamente uno più piccolo e uno più grande, senza ripetere quello che è scritto nella consegna del progetto, possiamo riassumere che è quindi una struttura dati gerarchica che permette di inserire, rimuovere e cercare valori in tempo proporzionale all'altezza dell'albero.

Implementazione generale:

Dal momento che i nodi dell'albero sono rappresentati da una chiave e da un valore, la classe BST ha tre template chiamati *key_type*, *value_type* e *comparator_type*.

I confronti vengono eseguiti sulle chiavi attraverso il terzo template *comparator_type* che è un *functor,* in cui è definito un metodo per confrontare elementi di tipo *key_type*, di default è *std::less<key_type>.*

Struttura del codice:

Il codice è stato strutturato in tre file: node.hpp, bst.hpp, iterator.hpp.

- **Node.hpp:**

Nel file node.hpp abbiamo una classe che rappresenta i nodi, che è una classe interna alla classe bst. Ogni nodo è definito come una *std::pair key_type, value_type*, e ogni nodo ha due puntatori, che rappresentano il nodo figlio sinistro e il nodo figlio destro. Sono stati inseriti anche dei puntatori all'elemento successivo, all'elemento precedente e all'elemento parent, che semplificano successivamente le operazioni con gli iteratori e le operazioni di inserimento, eliminazione e bilancio. Inoltre, oltre ai vari getters e setters, abbiamo l'operatore put-to e alcuni controlli che semplificano l'utilizzo del nodo, come per esempio *is_left*, che restituisce un valore booleano che indica se il nodo è un figlio sinistro, *is_root* che indica se il nodo è la radice, o *is_leaf* che indica se il nodo è una foglia.

- **Iterator.hpp:**

Nel file iterator.hpp c'è la classe che definisce gli oggetti *iterator* e *const_iterator,* anche questa è una classe interna a bst. I quali vengono inizializzati con un puntatore a nodo. Sono stati inseriti i vari overload degli operatori, in particolare l'operatore *++* che restituisce l'iteratore al nodo successivo del nodo puntato. È stata implementata anche una classe di iteratore costante per essere sicuri che nel momento in cui scorriamo l'albero non vengano modificati i valori dei nodi.

- **Bst.hpp:**

Questa è la classe principale, possiede un puntatore a nodo radice.

Il *copy constructor* di questa classe ovviamente esegue una deep copy che copia nodo per nodo a partire dalla radice.

È stato poi implementato un *clear()* che chiama il metodo *erase()* fintanto che la radice non è nulla.

Il metodo balance è stato implementato in maniera abbastanza naive, ma tuttavia è lineare, inserisce tutti i nodi in un array seguendo la visita inOrder (quindi partendo da *begin()*), svuota l'albero chiamando il metodo *clear()* e poi ricorsivamente inserisce l'elemento a metà array, poi chiama ricorsivamente il metodo sulla prima metà e sulla seconda metà dell'array. Sono stati poi definiti tutti gli operatori come da consegna.

- **Memory leak:**

Per verificare che non vi siano memory leak è stato utilizzato valgrind, attraverso l'istruzione:

valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes --verbose --log-file=valgrind-out.txt /main.exe

Il quale mostra che non ci sono memory leak e non vengono generati se si utilizzano le classi.

- **Per compilare:**

Per compilare il codice basterà inserire in una cartella le tre classi assieme al main, collocarsi nella cartella con il terminale e digitare:

"g++ main.cpp -Wall -Wextra -std=c++17 -o main.exe"

Per eseguirlo basterà poi scrivere "*./main.exe"*

- **Conclusione:**

Il progetto è stato portato a termine, i miglioramenti che possono essere fatti sono molteplici: per cominciare l'utilizzo degli smart pointer per migliorare la leggibilità e la scrittura del codice e per facilitare una gestione più delle risorse. Inoltre, la complessità del metodo *balance()* può essere sicuramente migliorata, magari utilizzando una struttura dati leggermente diversa, potrebbe essere chiamato dopo ogni *insert()* per garantire l'efficienza.