Esame di Programmazione C++

Nome: Davide
Cognome: Pozzoli
Matricola: 829584
email: d.pozzoli9@campus.unimib.it

Introduzione

Dopo un'accurata valutazione del problema, ho deciso di rappresentare l'albero binario di ricerca come una lista concatenata, formata da nodi contenti informazioni sui valori inseriti nell'albero e le relazioni padre e figlio tra i vari nodi. Oltre ad implementare i tradizionali puntatori per un albero binario di ricerca, figlio sinistro e figlio destro, ho aggiunto anche un puntatore *next*, che punta al nodo inserito nell'albero subito dopo in modo da facilitare l'implementazione dell'iteratore e della stampa dell'albero. Non ho trovato necessario l'utilizzo del puntatore al nodo padre e quindi non l'ho implementato.

Scelte implementative

L'albero viene implementato grazie alla classe *albero_binario*. Non essendo prevista la compilazione separata delle definizioni dei template e delle loro implementazioni l'intera classe é sia dichiarata che definita all'interno del file *alberobinario.hpp*.

La classe include 3 diversi template, il primo **T** identifica il tipo di valori inseriti nell'albero, gli altri due **D** ed **E** rappresentano i due funtori per la comparazione dei valori all'interno dell'albero, rispettivamente l'operatore '$<$' e l'operatore '$==$'.

Vengono anche create 2 classi di eccezioni custom che sono *duplicated_node_exception* che viene lanciata nel momento in cui si tenta di inserire un valore nell'albero giá presente e *node_not_found_exception* che viene lanciata nel momento in cui si tenta di ottenere dall'albero un nodo non presente.

Parte privata

Nella parte privata della classe troviamo, la struttura interna **node** che rappresenta i nodi di cui si compone l'albero. Al suo interno troviamo definiti 3 puntatori, oltre alla variabile value di tipo T. I tre puntatori sono rispettivamente al figlio sinistro del nodo a quello destro e al nodo inserito successivamente rispetto a quello. I metodi presenti nella struttura sono unicamente i vari tipi di costruttori e il distruttore.

Nella parte privata al di fuori della struttura di supporto troviamo il puntatore ad un nodo che rappresenterá la radice dell'albero e un unsigned int che

rappresenta il numero di nodi presenti all'interno dell'albero. Sono definiti in questa parte i due oggetti funtori per l'ordinamento e l'eguaglianza utilizzati successivamente per tenere in ordine l'albero.

Nella parte privata troviamo anche i tre metodi helper ricorsivi che servono rispettivamente per il metodo subtree e i due tipi di stampe, la prima eseguita in order, la seconda eseguita in modo da mostrare l'albero in una semplice rappresentazione grafica.

Parte pubblica

Nella parte pubblica troviamo innanzitutto i costruttori, di default e non, il disruttore e l'operatore di assegnamento. I due metodi successivi **get_num_nodi** (richiesto dalla traccia al punto 1) e **get_root** sono i due metodi get per le variabili rispettivamente *num_nodi* e *root*.

Successivamente vi sono i metodi **get_node** e **check_node**. Quest'ultimo é richiesto dalla traccia al punto 2 e serve a stabilire se il valore di tipo T é presente nell'albero ritornando un valore booleano. Il primo invece é un metodo che ho implementato per ottenere il nodo all'interno dell'albero corrispondente al valore di tipo T passato come parametro, fa uso di *check_node* al suo interno per per verificare la presenza del nodo e in caso negativo lancia l'eccezione *node_not_found_exception*. Questo metodo viene usato all'interno del metodo subtree definito piú avanti.

Il metodo **add** é il metodo che permette di inserire un nuovo nodo all'interno dell'albero binario mantenendolo di ricerca. La quasi totalitá del metodo é inserito all'interno di un blocco try per evitare problemi con l'allocazione di memoria che nel caso si verificassero sarebbero catturati da un eccezione generica. All'interno del blocco try é presente anche il throw di un'altra eccezione che viene lanciata nel momento in cui il metodo si accorge che il valore che si sta provando ad inserire é giá presente nell'albero. Entrambe le eccezioni non sono bloccanti ma si limitano a stampare il messaggio di errore ed eliminano l'oggetto node creato con la new.

Il metodo gestisce i puntatori dei nodi tenendo in ordine anche il puntatore next usato nell'iteratore e aumenta la variabile *num_nodi* successivamente all'inserimento.

Il metodo **subtree** é il metodo richiesto nel punto 5 della traccia e serve a restituire un nuovo albero formato a partire da un nodo presente all'interno del albero principale. Per ottenere questo risultato utilizza il metodo ricorsivo helper **subtree_helper** che si occupa della creazione del nuovo albero usando il metodo add. Per ottenere il nodo, *subtree* utilizza il metodo *get_node* che nel caso in cui non trovasse il nodo cercato lancia la *node_not_found_exception*. L'eccezione non é bloccante ma viene gestita con un blocco catch che stampa il messaggio di errore e ritorna un albero vuoto creandolo con il costruttore di base.

Successivamente troviamo l'**iteratore** costante richiesto dalla traccia al punto 3. L'iterazione avviene partendo dal nodo radice e spostandosi da un nodo all'altro seguendo il puntatore next e quindi l'ordine di inserimento. Subito dopo la definizione della classe troviamo anche i due metodi **begin** ed **end** necessari alla classe per utilizzare l'iteratore.

Gli ultimi 2 metodi sono **in_order** e **print_h**, che sono rispettivamente 2 metodi alternativi utilizzati per la stampa che vanno ad affiancarsi all'operatore « che invece utilizza l'iteratore per stampare la sequenza.

Metodi globali

I due metodi globali definiti all'interno della classe sono **l'operatore«** che viene ridefinito per la stampa dell'albero come richiesto nel punto 4 della traccia e il secondo é il metodo **printIF** che anche questo richiesto dalla traccia.

La ridefinizione dell'*operatore«* avviene sfruttando *l'iteratore*, quindi i nodi vengono stampati in ordine di inserimento, stessa metodo é utilizzato nel metodo *printIF* dove peró la stampa avviene solo nel caso si rispetti il predicato passato come parametro.

Qt

Per quando riguarda le scelte implementative in qt, ho deciso che nel momento in cui, successivamente al download della lista di artisti, uno di essi:

- abbia specificato solo il nome ed il link sia una stringa vuota, allora l'artista sará inserito con un link che porta alla pagina principale di wikipedia.
- non abbia specificato il nome allora in quel caso, viene aggiunta un label alla lista con al posto del nome la stringa "???" e nessun collegamento inserito. In questo caso si va ad aggiungere al grafico nella barra others che comprende tutti gli artisti che iniziano con qualcosa di diverso dalla lettera dell'alfabeto.

Purtroppo in Qt, nonostante i vari tentativi effettuati, non sono riuscito a risolvere un problema di memory leak rilevato da valgrind nel momento in cui viene cliccata la label per aprire il collegamento ipertestuale. Cercando di risolvere il problema mi sono imbattuto nel post pubblicato nel forum di Qt https://forum.qt.io/topic/112949/view-a-list-of-qlabel/3 che é la risposta piú vicina al problema riscontrato.

Mentre durante l'esecuzione normale posso aprire quante label voglio da entrambe le liste di artisti, durante l'esecuzione con valgrind dopo la terza label cliccata il programma va in freeze.