**Relazione Progetto Programmazione C++ 20 Aprile 2020**

Nome: Andrea

Cognome: Tassi

Matricola: 830541

Indirizzo email d’ateneo: [a.tassi1@campus.unimib.it](mailto:a.tassi1@campus.unimib.it)

Indirizzo email personale: [andreatassi98@gmail.com](mailto:andreatassi98@gmail.com)

**Introduzione**

Il progetto richiedeva la realizzazione di una classe generica che implementasse un albero binario di ricerca, formato da un insieme di elementi di tipo generico T contenuti in nodi connessi in una struttura gerarchica padre-figlio.

**Tipi di dati**

**class binary\_search\_tree<T, O, E>**

Per realizzare la classe che rappresenta l’albero (binary\_search\_tree), essendo i suoi elementi di tipo generico T, è stato necessario definirla come classe template, con tre parametri template:

1. T, che rappresenta proprio il tipo generico dei dati contenuti nei nodi dell’albero;
2. O, che rappresenta il funtore di confronto di ordinamento di due dati dell’albero;
3. E, che rappresenta il funtore di confronto di uguaglianza di due dati dell’albero.

Questi ultimi due parametri template sono stati necessari, in quanto doveva essere possibile per l’utente scegliere la strategia usata per confrontare due dati T: dato che il confronto può avvenire sia sull’ordine (operatore < sui tipi primitivi) sia sull’uguaglianza (operatore == sui tipi primitivi), allora necessitavo di altri due parametri template, per identificare, rispettivamente, la modalità di confronto sull’ordine e quella sull’uguaglianza.

**struct node**

Per rappresentare i nodi, ho deciso di definire, all’interno della classe stessa, una struct privata che rappresentasse un generico nodo dell’albero, con il valore contenuto al suo interno definito come dato membro privato di tipo T.

Inoltre, per via della struttura gerarchica padre-figlio, era necessario che ogni nodo contenesse un’informazione sui suoi figli: quest’informazione è contenuta nei suoi dati membro left e right, puntatori, rispettivamente, al figlio sinistro e a quello destro.

Tuttavia, è stato necessario, soprattutto per le operazioni di visita degli iteratori (descritte più avanti), poter risalire al padre di un certo nodo: perciò, ho incluso un altro dato membro privato, parent, un puntatore al nodo padre.

I costruttori della struct sono tre:

1. Default: per istanziare un nodo vuoto, che non ha né valore, né padre, né figli.
2. Secondario: per istanziare un nodo con il suo valore, ma senza padre né figli.
3. Secondario: per istanziare un nodo con valore e puntatori a padre e figli.

Il distruttore, invece, permette di rimuovere la memoria allocata da un nodo, ponendo i tre puntatori a nullptr.

N.B: non ho deciso di disallocare i nodi puntati dai tre puntatori, in quanto potrebbero essere puntati da altri nodi ancora in fase di utilizzo (un nodo, infatti, oltre a puntare ad altri tre nodi, è a sua volta puntato da tre nodi, ovvero i suoi figli e il suo padre). Questo potrebbe generare errori nei puntatori dei nodi che puntavano al nodo eliminato, che ora punterebbero ad un dato non più presente in memoria, ma questo problema è evitato dal fatto che un nodo non viene mai eliminato singolarmente, ma solo insieme a tutti gli altri nodi dell’albero di cui fa parte, evitando così memory leak. In poche parole: se viene eliminato un nodo, allora è perché si sta eliminando l’intero albero, e, viceversa, se viene eliminato un intero albero, allora vengono eliminati tutti i suoi nodi.

**Dati membro privati**

I dati membro privati della classe binary\_search\_tree sono 4:

1. \_root, un puntatore alla radice (per poter poi accedere, tramite essa, a tutti gli altri nodi);
2. \_size, di tipo size\_type (ovvero un unsigned int), per memorizzare il numero totale di elementi inseriti nell’albero (non è strettamente necessario per il funzionamento dei metodi principali della classe, tranne che per il metodo size() che ne ritorna, appunto, il valore, come richiesto dalla traccia);
3. \_order, un oggetto funtore di tipo O (parametro template) per il confronto di ordinamento;
4. \_equals, un oggetto funtore di tipo E (parametro template) per il confronto di uguaglianza.

**Funzioni helper private**

Ho incontrato la necessità di dichiarare e definire 4 funzioni helper private per il corretto funzionamento sia dei metodi fondamentali, sia di quelli d’uso dell’interfaccia pubblica, e per evitare code smells come codice duplicato, dato che sono funzioni che vengono riutilizzate da più funzioni e metodi:

1. insert\_tree, che inserisce ricorsivamente gli elementi di un albero (a partire dal puntatore alla sua radice) in quello corrente (utile soprattutto per il Copy Constructor): gli elementi vengono inseriti in pre-order;
2. clear, che elimina l’intero contenuto dell’albero (utile soprattutto per il distruttore), richiamando la funzione helper ricorsiva clear\_tree, a partire dalla radice. È stato necessario definirla come funzione a parte, e non direttamente nel distruttore, in quanto anche il codice del Copy Constructor e quello di subtree, in caso di errori di allocazione di memoria, utilizzano questa funzione (per evitare di richiamare esplicitamente il distruttore).
3. clear\_tree, che disalloca ricorsivamente i nodi dell’albero a partire dalla radice: gli elementi vengono eliminati in post-order, per evitare memory leak;
4. search, che cerca se esiste nell’albero un nodo con un certo valore e, in caso affermativo, ne ritorna un puntatore (l’uguaglianza è definita dal funtore \_equals di tipo E dell’albero). La visita avviene a partire dalla radice e confrontando di volta in volta il valore del nodo corrente e quello da trovare: se quest’ultimo è minore, si confronta il figlio sinistro, se è maggiore il destro, altrimenti, se sono uguali, la ricerca è completata.

N.B: Sebbene dovessi già implementare un’altra funzione simile (exists), con solo tipo di ritorno diverso (bool), ho avuto bisogno di definire questa funzione helper per permettere al metodo subtree di ottenere subito la radice da cui costruire il sottoalbero. A quel punto, ho trovato conveniente riutilizzare questa porzione di codice di ricerca anche per la exists, ed ecco il motivo della definizione della funzione privata search.

**Metodi fondamentali**

1. Costruttore di default: ho deciso di definire solo il costruttore di default, in quanto l’inserimento di nodi nell’albero avviene singolarmente o tramite cicli a partire da una lista di valori già esistente, ma, in ogni caso, sempre un elemento alla volta, tramite chiamata al metodo pubblico insert. Inoltre, è l’unico costruttore che può essere utilizzato per istanziare un eventuale array di alberi (utilizzato, infatti, nei metodi di test della classe main);
2. Copy Constructor: permette di istanziare un albero con i valori presi da un altro albero. L’inserimento nell’albero corrente avviene grazie alla funzione helper privata insert\_tree, a partire dalla radice dell’altro albero. Se viene sollevata un’eccezione di allocazione di memoria, prima di rilanciarla al chiamante, viene disallocato tutto l’albero corrente finora allocato (tramite chiamata alla funzione helper privata clear), in modo da evitare memory leak;
3. Operatore di assegnamento (=): permette la copia tra alberi. Per evitare l’auto-assegnamento (tree = tree), ho dovuto porre a guardia del codice un if-statement, per verificare che i due alberi (sorgente e destinatario) non siano effettivamente lo stesso albero (non con gli stessi valori, ma proprio lo stesso oggetto);
4. Distruttore: rimuove la memoria allocata dall’albero, tramite chiamata alla funzione helper privata clear (già descritta precedentemente).

**Metodi d’uso dell’interfaccia pubblica**

1. insert: inserimento di un valore nell’albero. L’inserimento avviene verificando l’ordinamento del valore da inserire con quelli esistenti, a partire dalla radice (l’ordinamento è definito dal funtore \_order, di tipo O (parametro template)): se il valore da inserire è minore di quello corrente, viene confrontato con il figlio sinistro, altrimenti con quello destro. Se non ci sono più figli (il nodo corrente è un nodo foglia), il nuovo valore viene inserito nella posizione in cui ci sarebbe dovuto essere l’ulteriore figlio da confrontare (figlio sinistro se il valore è minore del nodo foglia, figlio destro se è maggiore). Viene istanziato nell’heap un nodo temporaneo con il nuovo valore, per venire poi assegnato alla posizione corretta, aggiornando il puntatore del parent al figlio (cioè al nuovo nodo) e il puntatore al parent del nodo stesso. Per assicurare la constness (const correctness), questo metodo non è definito const, in quanto modifica l’albero corrente (inserendo un nodo e modificando un puntatore di un nodo pre-esistente).

N.B: se il valore da inserire risulta uguale a uno di quelli già esistenti nell’albero, viene lanciata un’eccezione (bst\_duplicated\_value\_exception), in quanto dev’essere impedito l’inserimento di dati duplicati. L’eccezione lanciata contiene tutte le informazioni necessarie a comprenderne il motivo (messaggio visualizzabile tramite metodo what() e valore duplicato visualizzabile tramite metodo get\_duplicated\_value());

1. size: ritorna il numero totale di elementi inseriti nell’albero (dato membro \_size). Per assicurare la constness, questo metodo è definito const, in quanto assicura la non modifica dell’albero corrente (deve solo restituire il valore di un suo dato membro);
2. exists: controlla se esiste un nodo con un certo valore, passato come parametro. Avviene tramite chiamata alla funzione helper search() e verificando che il puntatore restituito non sia nullptr;
3. subtree: passato un dato d dello stesso tipo T dei dati contenuti nell’albero, ritorna un nuovo albero, che corrisponde al sottoalbero avente come radice il nodo con il valore d. Innanzitutto, tramite la funzione helper search(), acquisisce il puntatore alla nodo con valore d: se il valore non esiste, viene lanciata un’eccezione (bst\_value\_not\_found\_exception), in quanto non è possibile costruire un sottoalbero da un valore non esistente. L’alternativa era restituire un sottoalbero vuoto, ma ho preferito considerare insensata questa strategia, in quanto viene richiesto il sottoalbero con un certo valore come radice: se questo non è presente, è impossibile costruirlo, da cui la scelta dell’eccezione, che contiene tutte le informazioni necessarie a comprenderne il motivo (messaggio visualizzabile tramite metodo what() e valore non presente visualizzabile tramite metodo get\_not\_found\_value());