Relazione Progetto C++
Aprile 2020

Nome: Daniel
Cognome: Furfaro
Matricola: 810933
e-Mail: d.furfaro1@campus.unimib.it

Progetto:

Il progetto consiste nella realizzazione di una classe che implementa un albero binario di ricerca, per la sua realizzazione è stata utilizzata la seguente versione: GCC Build-20200227-1 9.2.0

Tipi di dato:

L'albero è composto da Nodi connessi tra loro con relazioni di padre (nodo con livello di profondità -1 dal nodo preso in esame, se tale non è la radice) e 0,1 o 2 figli (sinistro e destro, con profondità +1 dal nodo preso in esame).

L'albero (`class binary_tree`) contiene i seguenti dati:

- `root`: Nodo radice (unico) e da cui parte ogni altro nodo sottostante
- `funct`: Predicato in funzione del quale l'albero verrà ordinato

Ogni Nodo (una "struct") contiene i seguenti dati:

- `value`: un dato di tipo T (generico), che rappresenta il valore contenuto nel nodo/foglia.
- `father`: un puntatore al nodo padre del nodo preso in esame (null nel caso della radice)
- `left`: un puntatore al nodo figlio sinistro (può essere null)
- `right`: un puntatore al nodo figlio destro (può essere null)

Il tipo di dato Node è dotato di tre costruttori:

- `node/0`: utile nel momento in cui viene creato un albero con radice vuota.
- `Node/2`: utile alla creazione di un nodo/foglia senza figli.
- `Node/4`: utile alla creazione di un nodo intermedio (con padre e figli)

Implementazione Albero:

Come in ogni classe sono stati implementati il costruttore di default, costruttori secondari, operatori fondamentali e il distruttore.

Il costruttore di default (`binary_tree/0`) crea un albero con radice vuota.

Sono inoltre presenti due costruttori secondari:

- `binary_tree/1`: Un costruttore che assegna alla radice il valore passato
- `binary_tree(tree)/1`: Un costruttore che crea una copia dell'albero passato come parametro

Come esplicitamente richiesto nella consegna, due operatori:

- `operator<<`: Che passato un albero, si occupa, tramite l'utilizzo di iteratori a stampare la struttura dell'albero.
- `operator=`: Che restituisce un albero copia dell'albero passato (A = B).

e infine il distruttore che appoggiandosi ad una funzione secondaria `clear_helper`, in modo ricorsivo si occupa della "pulizzia" dei puntatori `father, left e right` eseguendo in fine una `delete` sul nodo stesso.

Metodi Implementati:

Aperto il codice il primo metodo che troviamo è il `search_node/1`, inizialmente non avrei implementato questo metodo, poiché il suo contenuto poteva essere semplicemente inglobato nel metodo `search/1` espressamente richiesto nella consegna, ma nel momento in cui mi sono trovato di fronte all'implementazione di `subtree/1` ho notato che stavo riscrivendo un codice già (precedentemente) scritto per la funzione, ho quindi optato per la creazione di questo metodo di supporto che restituisce l'indirizzo del nodo cercato, che nel caso della funzione `search` viene logicamente riconvertito in un `bool` e ritornato per il responso sulla ricerca.

Dopo di che troviamo `copy_tree_by_node/1 e copy_tree/1` altri due metodi di supporto, che, anche in questo caso svolgono un compito richiesto in altri metodi "principali", quali il costruttore `binary_tree(tree), subtree/1 e operator=`, che necessitano della restitituzione e conseguente copia di alberi partendo da nodi diversi (non sempre radici).

Proseguendo troviamo `clear_helper/0 e rec_node_del/1`, il primo è il metodo principale per la pulizia dell'albero che si limita a richiamare il secondo per avviare la ricorsione.

In seguito abbiamo `size/1 (privato) e unsint get_size/0 (publico)`, il secondo si limita a richiamare il primo il quale si occupa partendo dalla radice (ricorsivamente) di calcolare la quantità di nodi presenti nell'albero.

Subito dopo ci sono i già citati `search e subtree`, che si limitano ad utilizzare i già descritti sopra `search_node` (nel caso di `search`), e `search_node + copy_tree` (nel caso di `subtree`).
L' `operator=` si limita a richiamare `clear_helper`, per evitare problemi nella conseguente copia, e `copy_tree` per effettuare la copia del albero passato su `this->root (l'albero su cui viene utilizzato l'operatore)`.

Arriviamo ora ad un altro metodo principale, `add/1`, qualòra la radice fosse vuota questo si limiterà ad inserire il valore e ritornare, altrimenti ho deciso anche in questo caso di utilizzare un metodo ricorsivo appoggiandomi quindi sulla funzione `find_position/2`, come specificato nella consegna se all'inserimento viene individuato un valore uguale a quello passato, il procedimento viene subito annullato, altrimenti in base al predicato `funct`, continuo a "muovermi" a destra e sinistra rispettivamente nel caso in cui il valore passato sia maggiore o minore di quello presente nel nodo esaminato, fermandomi quando verrà individuato un nodo vuoto.

Entriamo ora nella parte riguardante gli Iteratori (`const_iterator`) (`std::forward_iterator_tag`), troviamo i costruttori e operatori fondamentali, `const_iterator/0,const_iterator/1, begin/0 , end/0, operator=, operator==, operator->, operator*` e soprattutto `const_iterator operator++/1 e const_iterator& operator++/0` l'idea iniziale era quella di implementare un operatore di incremento tale da ottenere una vista post-order, ma man mano che andavo avanti nella stesura del codice notavo che senza l'ausilio di strutture dati d'appoggio questo approccio risultava macchinoso e complesso, ho quindi deciso di passare ad un approccio pre-order cosiffatto, se l'iteratore punta a `nullptr` lancio un `throw`, altrimenti controllo se

il figlio sinistro e destro (in quest'ordine) puntano a `nullptr` se non lo fanno "mi muovo" in quella direzione, se in entrambi, ciò non si verifica vuol dire che si tratta di un nodo foglia (senza figli), dobbiamo quindi risalire l'albero per continuare ad attraversarlo, è però necessario memorizzare e capire da dove siamo arrivati, così da evitare di finire in un loop e tornare su nodi già visitati, salviamo quindi il nodo padre e tramite un ciclo cerchiamo di spostarci ordinatamente verso destra, dato che precedentemente abbiamo dato priorità dall'alto verso il basso e da destra a sinistra, quando verrà trovato un nodo con figlio destro e diverso dall'nodo di provenienza, questo verrà ritornato, altrimenti proseguo con il ciclo fino al punto di trovare un nodo senza padre (quindi saremo tornati alla radice), il quale padre punterà a `nullptr`, concludendo l'iterazione.

PS: Dato il tempo e il lavoro speso ho voluto conservare la parte di codice precedentemente scritto nel tentativo di creare una vista post-order, ovviamente lasciato come commento, era effettivamente funzionante, ma non ho trovato un metodo pratico per evitare un loop nel momento in cui si tornava al nodo radice.

In fine abbiamo `operator<< e printIF/2`, che utilizzando gli iteratori e l'incremento precedentemente esposto, si limitano tramite un ciclo a stampare i valori contenuti nell'albero, in aggiunta `printIF` per svolgere tale operazione controllerà che i valori esaminati rispettino un predicato/condizione passata dall'utente (`P pred`).