Relazione progetto C++
Aprile 2020

Nome: Davide

Cognome: Pietrasanta

Matricola: 844824

Mail: d.pietrasanta@campus.unimib.it

Tempistiche: Dal 20/03/2020 al 10/04/2020

----------< Introduzione >----------

Dopo un'accurata valutazione del problema, ho deciso di rappresentare l'albero binario tramite una classe BTree che rappresenta un insieme di nodi. Per separare l'entità nodo dall'albero stesso ho creato una struttura dati per i nodi e una classe per gli alberi, aventi entrambe quattro valori.

----------< Tipi di dati >----------

Ho deciso, come detto poc'anzi, di separare nodi e alberi. Ho dunque creato una struttura dati **Node** per descrivere un nodo di un albero binario. Essa contiene quattro attributi:

- **value** è il valore di tipo generico T scelto dall'utente in fase di inizializzazione
- **right** è il puntatore al nodo destro
- **left** è il puntatore al nodo sinistro
- **next** è il puntatore al prossimo nodo, per gli iteratori

Tutti gli attributi di Node, oltre che Node stessa, sono pubblici in modo da consentire all'utente un uso ristretto anche solo ai nodi.

La classe albero, chiamata **Btree**, contiene quattro attributi:

- **sz** è un unsigned int che contiene la dimensione dell'albero
- **root** è il puntatore al nodo che fa da radice dell'albero
- **last** è il puntatore all'ultimo next dell'albero
- **funct** è di tipo F, cioè l'operatore di confronto scelto dall'utente in fase di inizializzazione

Separando l’entità albero da quella nodo abbiamo un unico valore a rappresentare la grandezza dell’albero, così da avere un ottimizzazione della memoria.

Tutti gli attributi di Btree sono privati in modo da non consentire la manipolazione non controllata dell’albero.

----------< Implementazione >----------

Come espressamente richiesto dalla consegna, non era possibile l’utilizzo di librerie esterne e strutture dati container della std library, ho dunque implementato le funzioni standard per la struttura Node e la classe Btree.

Per la classe Btree sono stati definiti:

- **Costruttore di default**, che inizializza un albero vuoto
- **Costruttore copia**, che inizializza un albero copia di un altro
- **Costruttori secondari**
  - Per inizializzare tramite un generico valore di tipo T, ovvero inizializza un albero di dimensione uno e con valore della radice pari al valore inserito.
  - Per inizializzare tramite un Nodo, ovvero inizializza un albero che è una copia del nodo e dell’insieme di nodi ad esso collegati.
- **Distruttore**, che elimina ogni allocazione effettuata in memoria durante il corso dello svolgimento del programma utilizzando una funzione di supporto *destroy()*
- **operator =**, che permette assegnamento

Per la struttura Node sono stati definiti:

- **Costruttore di default**, che inizializza un nodo con valore 0
- **Costruttore copia**, che inizializza un nodo copia di un altro e dell’insieme di nodi ad esso collegati
- **Costruttori secondari**
  - Per inizializzare tramite un generico valore di tipo T, ovvero inizializza un nodo con valore pari al valore inserito e con puntatori next, right e left “nullptr”.
  - Per inizializzare tramite un puntatore a un nodo.

Ho ritenuto meglio dichiarare il funtore di confronto F all’inizializzazione dell’albero. In questo modo si può garantire un albero consistente, avendo un solo tipo di inserimento. Il funtore di default ordina i valori in modo crescente. F deve aver definito *operator()*. Inoltre dato funct di tipo F, allora funct(A,B) deve tornare 0 se A e' uguale a B, 1 se A e' minore di B e -1 se A e' maggiore di B.

----------< Metodi implementati >----------

Nella classe Btree:

- **size()** - E' stata richiesta una funzione per conoscere il numero totale di dati nell'albero, ho dunque implementato la funzione size() che ritorna il valore dell'attributo sz presente nell'oggetto. Alla creazione viene inizializzato e ad ogni insert() il valore viene aggiornato. Nel caso di assegnamento, inizializzazione per costruttore copia o inizializzazione per costruttore con nodo esso viene ricalcolato con l'ausilio della funzione size() presente in Node. Ho deciso di creare un attributo apposito per la dimensione dell'albero in modo da velocizzare la funzione rendendola di complessità temporale costante.
- ***Getter*** – Ho implementato dei metodi getter per i valori della root, ovvero **get_root()**, e della last, ovvero **get_last()**. Inoltre ho implementato le funzioni **max()** e **min()** che ritornano il valore, rispettivamente, del massimo e del minimo nell'albero rispettando il funtore di confronto scelto dall'utente in fase di inizializzazione.
- **Insert(const T)** - Permette l'inserimento di un valore di tipo T. Se il valore è duplicato non lo inserisce, se il valore non è invece presente nell'albero crea un nodo con quel valore, lo inserisce in modo ordinato (rispettando il funtore di confronto) e incrementa la dimensione dell'albero. Utilizza una funzione privata *insert(const T, Node*)* ricorsiva, essa ha il compito di inserire in modo ordinato il nodo, il tutto con complessità temporale media logaritmica (se l'albero non è bilanciato la complessità potrebbe anche essere lineare).
- ***Ricerca –*** Deve esser possibile il controllo di esistenza di un elemento T. Ho dunque implementato due funzioni. **search(const T)** cerca il un elemento di tipo template T nell'albero seguendo l'ordinamento scelto dall'utente tramite il funtore di confronto. Ritorna nullptr se il valore non è presente, altrimenti ritorna una copia del nodo (In questo modo si conserva la sicurezza).Utilizza una funzione privata *search(const T, Node*)* . La funzione **has(cons T)** sfrutta la funzione *search(const T)* per scoprire se un valore è presente o meno nell'albero e ritorna un booleano. Il tutto viene compiuto con complessità temporale media logaritmica (se l'albero non è bilanciato la complessità potrebbe anche essere lineare).
- **destroy()** - Usa l'ausilio di *destroy(Node*)* per liberare la memoria allocata. Lo fa in modo ricorsivo e decrementa la dimensione (*sz*) dell'albero. Viene anche usata dal distruttore. Il tutto viete attuato con complessità temporale lineare.
- **subtree(const T)** – Costruisce un sottoalbero (che è ancora un albero) . Se l'albero è vuoto o il valore inserito non è presente nell'albero allora il sottoalbero sarà un albero vuoto. Se il valore è presente crea un albero che è una copia avente come radice quel valore.
- **Stampa –** E' possibile visualizzare l'albero tramite la funzione **stampa()** che utilizza la funzione *stampa()* della struttura *Node,* che opera in modo ricorsivo. Se l'albero è vuoto viene segnalato e la stampa avviene tramite std::cout. Ho inoltre ridefinito **operator<<** che utilizza gli iteratori. Anche in questo caso se l'albero è vuoto viene segnalato. Le due stampe posso avere un ordine diverso di

visualizzazione dell'albero. Infine è stata richiesta una funzione templata globale **printIF(Btree<T,F>& , P)**. Essa prende in ingresso un albero con valori di tipo T e funtore di confronto di tipo F e stampa solo gli elementi dell'albero che soddisfano il predicato booleano P. Anch'essa utilizza gli iteratori. Tutte le funzioni di stampa sono attuate con complessità temporale lineare.

Nella struttura Node:

- **insert_node(T)** - Permette l'inserimento di un valore di tipo T. Se il valore è duplicato non lo inserisce, se il valore non è invece presente crea un nodo con quel valore, lo inserisce in modo ordinato (rispettando il funtore di confronto). E' una funzione ricorsiva e operata con complessità temporale media logaritmica (se l'albero non è bilanciato la complessità potrebbe anche essere lineare).
- **copy_node()** - Permette la copia di un nodo. Opera usando *copy_node_ric()* che a sua volta utilizza la funzione ***array_node()***. Quest'ultima riempie in modo ricorsivo un array di puntatori a Node. Alla fine dell'esecuzione nel puntatore ci sarà un elenco dei nodi di un albero. Si eseguono una serie di insert_node scorrendo nell'array. In questo modo si ritorna il nodo copia. Dopo di ciò si chiama *force_next()* per forzare il ricalcolo dei next e garantirne una consistenza. Il tutto avviene con complessità temporale asintoticamente lineare.
- **Size()** - Calcola in modo ricorsivo la grandezza dell'insieme di nodi. La funzione ha complessità temporale lineare.
- **get_value()** - Metodo getter per il value di un Node.
- **stampa()** - Stampa in modo ricorsivo l'insieme di nodi partendo dalla radice. In caso di Nodo nullptr lo segnala. La funzione ha complessità temporale lineare.
- **last_next()** - Ritorna l'ultimo next dell'insieme di nodi scorrendo l'albero attraverso i puntatori next.
- **destroy()** - Libera tutta la memoria allocata del nodo, opera in modo molto simile a *destroy(Node*)* . Entrambe le funzioni sono ricorsive e richiamano se stesse finché tutta la memoria allocata non è stata liberata. Entrambe le funzioni hanno complessità temporale lineare. ATTENZIONE: Se si dichiara un nodo *Node * n = new Node(...)* si deve poi liberare la memoria esplicitamente utilizzando *n.destroy()*.

Nota bene: Se si vuole costruire un albero è meglio ignorare la classe Node e lavorare direttamente sulla classe Btree. Node è stata resa pubblica solo per garantire un possibile modellazione futura da parte dell'utente.

----------< Iteratori >----------

La specifica di progetto richiedeva l'implementazione di iteratori a sola lettura e di tipo forward con ordine di visualizzazione non rilevante. Come da richiesta è stato implementato un const_iterator. Per scorrere l'albero utilizza i puntatori next presenti in Node. La insert opera sui next collegando l'ultimo next (last) al nuovo elemento inserito.

Durante la subtree questo ordine viene ricalcolato tramite una funzione che ne forza la ricostruzione (*force_next()*).

----------< Main >----------

Nel file main.cpp vengono testati tutti i metodi pubblici resi disponibili dalla classe Btree e dalla struttura Node. Vengono testate diverse modalità di utilizzo dei metodi e tramite assert viene verificata la correttezza dei risultati. Come richiesto non è quindi necessario nessun input da parte dell'utente.

Vengono testati gli iteratori e la loro eccezione custom.

Vengono inoltre testate tutte le eccezioni custom. Ho ritenuto necessario crearne due:

- **uninitialized_value** serve per controllare l'accesso a valori non ancora inizializzati.
- **out_of_range_iterator** serve per controllare l'accesso ad indici dell'iteratore maggiori di end().

Ho provveduto anche a testare inizializzazioni della classe passando tipi diversi. Ho effettuato test con alberti di tipo int, char, string e una classe custom. La classe custom deve esser provvista di operatori < , > , == e << .

Ho inoltre ho testato diversi funtori custom per il confronto.