Esame di Programmazione C++ 20/04/20

Nome	Jacopo
Cognome	Maltagliati
Matricola	830110
e-Mail	j.maltagliati@campus.unimib.it

Scopo del Progetto

Implementare, come da specifica, una classe templata che rappresenti un albero binario di ricerca in C++, seguendo lo standard C++03 con alcune eccezioni.

Note Implementative

Per implementare l'albero binario di ricerca è stata utilizzata una struttura basata su nodi (`btree::node`). I nodi sono una struttura dati contenente un campo dati (`btree::node::_data`) di tipo `T` ed una serie di puntatori a nodi. I puntatori sono necessari per collegare i nodi tra loro, in modo che essi siano collegati tra loro a formare due strutture dati distinte:

- Un albero binario di ricerca
- Una coda coda singolo-linkata

Il dato contenuto nel nodo è dichiarato costante, in modo che non sia possibile manipolarlo dopo l'inserimento.

L'albero binario di ricerca

La prima stuttura di collegamento definita dai nodi è un albero binario, implementato tramite l'associazione ad ogni nodo di due puntatori: `btree::node::_left` e `btree::node::_right`, i quali assumono valore `nullptr` nel caso il nodo non abbia un figlio in quella direzione, e valore `node*` in caso contario. L'accesso alla struttura dati avviene in maniera ricorsiva, seguendo uno schema affine al seguente frammento di codice:

```
void r_visit(node* cur_node) {
```

```
        if(cur_node == nullptr) { return; }
        // fai qualcosa
        r_visit(cur_node->_left);
        r_visit(cur_node->_right);
    }
```

Questa modalità generica di accesso viene impiegata, in varie declinazioni, in quasi tutti i metodi della classe `btree`, al fine di implementare l'accesso, la copia, la ricerca e la stampa dei nodi in ordine logico.

Al fine di permettere che la stampa della struttura dati possegga certe caratteristiche, inoltre, è presente un puntatore al genitore del nodo (`btree::node::_parent`).

All'interno della classe templata `btree`, inoltre, è presente un puntatore a nodo detto `btree::_root`, che indica la radice dell'albero l-r. Ciò permette ai vari metodi della classe di avere un punto di partenza da cui effettuare l'accesso all'intero albero.

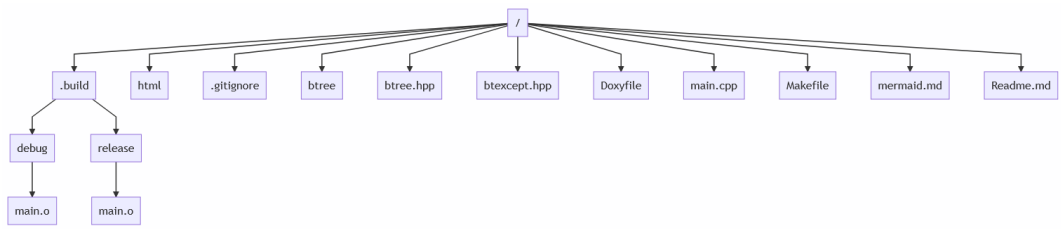
La coda singolo-linkata

Al fine di rendere possibile l'implementazione di un iteratore, inoltre, è stata utilizzata una seconda struttura dati, ovvero una coda singolo-linkata. Ogni nodo possiede infatti un puntatore (`btree::node::_qnext`), che indica il prossimo nodo nella coda. Nel caso esso non esista, questo puntatore assume valore `nullptr`, e valore `node*` in caso contrario.

Questa struttura dati secondaria permette di vedere i nodi in ordine di inserimento, ed essendo "piatta", ovvero potendo essere indicizzata in una sola direzione, permette di attraversare l'"albero" in maniera iterativa.

All'interno della classe templata `btree`, inoltre, è presente un puntatore a nodo detto `_qlast`, che indica l'ultimo elemento della coda. Ciò permette di effettuare implicitamente un'operazione di accodamento, aggiungendo un nodo come `_qnext` del `_qlast` e aggiornando `_qlast`.

Struttura del Progetto



Glossario

- `/` - radice
  - `.build/` - output intermedio di compilazione
    - `debug` - per il target Debug
      - `main.o` - file oggetto
    - `release` - per il target Release
      - `main.o` - file oggetto
  - `html/` - output di Doxygen
  - `.gitignore` - configurazione di Git
  - `btree` - file eseguibile di output
  - `btree.hpp` - classe templata
  - `btexcept.hpp` - eccezioni custom
  - `Doxyfile` - configurazione di Doxygen
  - `main.cpp` - programma dimostrativo
  - `Makefile` - configurazione di GNU Make
  - `Readme.md` - sorgente della relazione allegata
  - `Readme.pdf` - relazione allegata

Segue una descrizione più accurata delle sezioni più rilevanti.

File `btree.hpp`

Il file `btree.hpp` implementa la classe templata `btree`, che rappresenta un albero binario di ricerca basato su tipi custom. Il programmatore che voglia avvalersi delle funzioni della classe deve specificare:

- Il tipo di dati contenuti nell'albero
- Una strategia di comparazione (maggiore/minore) sotto forma di funtore
- Una strategia di comparazione (uguaglianza) sotto forma di funtore

A questo punto, per creare un oggetto `btree` è possibile avvalersi, sostanzialmente, di tre metodi:

- Un costruttore base a partire da un dato
- Un costruttore di copia
- L'operatore di assegnamento

Il costruttore base genera un'eccezione nel caso si verifichi un'errore nell'allocazione della memoria.

> ***NOTA***
> Per impedire la creazione di un albero e di un nodo vuoti, non sono stati imimplementati costruttori di default: è necessario specificare

> sempre il contenuto di un nodo mentre si costruiscono l'albero e i nodi associati.

Una volta creato un oggetto di tipo `btree`, è possibile utilizzarne tutti i metodi dell'interfaccia pubblica, di cui segue una breve descrizione:

`btree::add()`

Permette di aggiungere un nodo all'albero binario, dato il suo contenuto. Si appoggia alle funzioni `btree::next_branch()`, per stabilire in quale direzione è opportuno inserire il nuovo nodo, e `btree::create_node()`, per creare effettivamente il nodo. Genera un'eccezione nel caso si tenti di aggungere un dato già presente, oppure nel caso si verifichi un'errore nell'allocazione della memoria.

`btree::get_size()`

Restituisce la dimensione attuale dell'albero binario di ricerca, restituendo il parametro `btree::_size`, che viene incrementato ogni volta che si aggiunge un nodo.

`btree::exists()`

Stabilisce, dato un possibile contenuto, se nell'albero esiste un nodo che lo contenga. Si appoggia a `btree::r_find_node()`, controllando che non restituisca un puntatore nullo.

`btree::print()`

Stampa, utilizzando l'overload dell'operatore di stream (`operator<<()`), l'albero binario di ricerca.

> ***NOTA***
> Si prega di leggere la sezione sulle note stilistiche riguardo alla sintassi di stampa.

`btree::subtree()`

Restituisce un sotto-albero a partire da un nodo, appoggiandosi a `btree::sub_copy()`, che permette di copiare ricorsivamente (tramite `btree::r_copy()`) i nodi a partire da un punto specifico. Genera un'eccezione nel caso in cui il nodo di partenza non sia presente nell'albero.

`operator<<()`

Overload dell'operatore di stream, stampa l'albero su uno stream appoggiandosi a `btree::stream_print()`.

> ***NOTA***
> Si prega di leggere la sezione sulle note stilistiche riguardo alla sintassi di stampa.

Sono inoltre a disposizione, come da specifica:

- Un iteratore costante forward standard, che permette di scorrere la coda associata all'albero
- Una funzione globale `printIF()`, che stampa tramite il suddetto iteratore i valori nella coda associata all'albero se e solo se soddisfano un predicato

Una descrizione più accurata delle funzioni della classe templata è disponibile nella documentazione Doxygen che può essere generata a partire dal codice sorgente e dal *Doxyfile*.

File `btexcept.hpp`

Il file `btexcept.hpp` implementa le eccezioni custom, effettuando l'overload del metodo `what()`, per restituire un messaggio di errore personalizzato.

Una descrizione migliore delle funzioni di ogni eccezione è disponibile nella documentazione Doxygen che può essere generata a partire dal codice sorgente e dal *Doxyfile*.

File `Doxyfile`

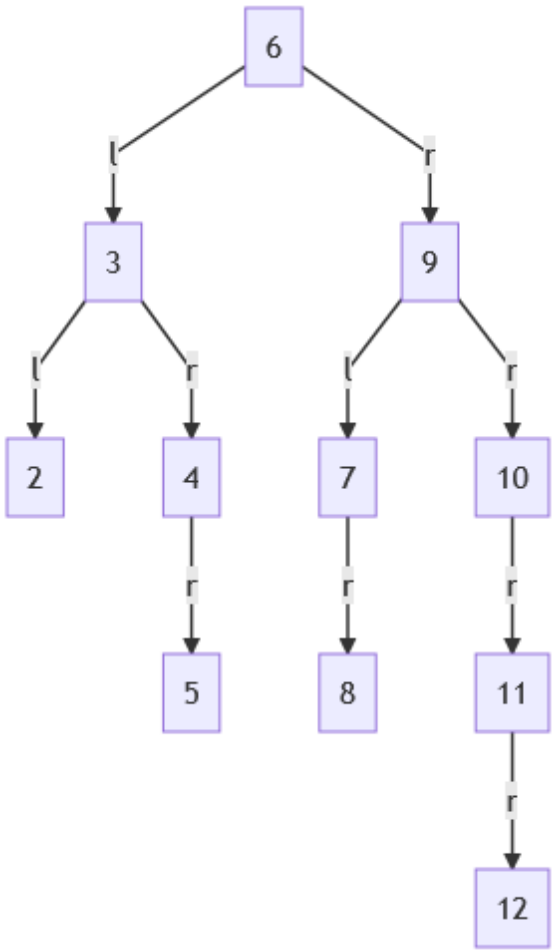
Il *Doxyfile* è stato personalizzando seguendo il modello già presentato durante il corso, ed introducendo varie modifiche aggiuntive: la presentazione della seguente, ad esempio, è resa possibile tramite l'introduzione della variabile `USE_MDFILE_AS_MAINPAGE`, che permette di utilizzare un file Markdown come pagina principale.

File `main.hpp`

Il file `main.hpp` implementa un semplice programma dimostrativo delle funzionalità della classe templata, che esegue due test sull'interfaccia pubblica della classe `btree`.

Il test sugli interi

Per prima cosa, vengono creati dei funtori per permettere la comparazione e la determinazione di uguaglianza tra due interi, e per permettere la determinazione della parità di un intero. Successivamente vengono aggiunti all'albero `bst_a` i nodi come in figura:

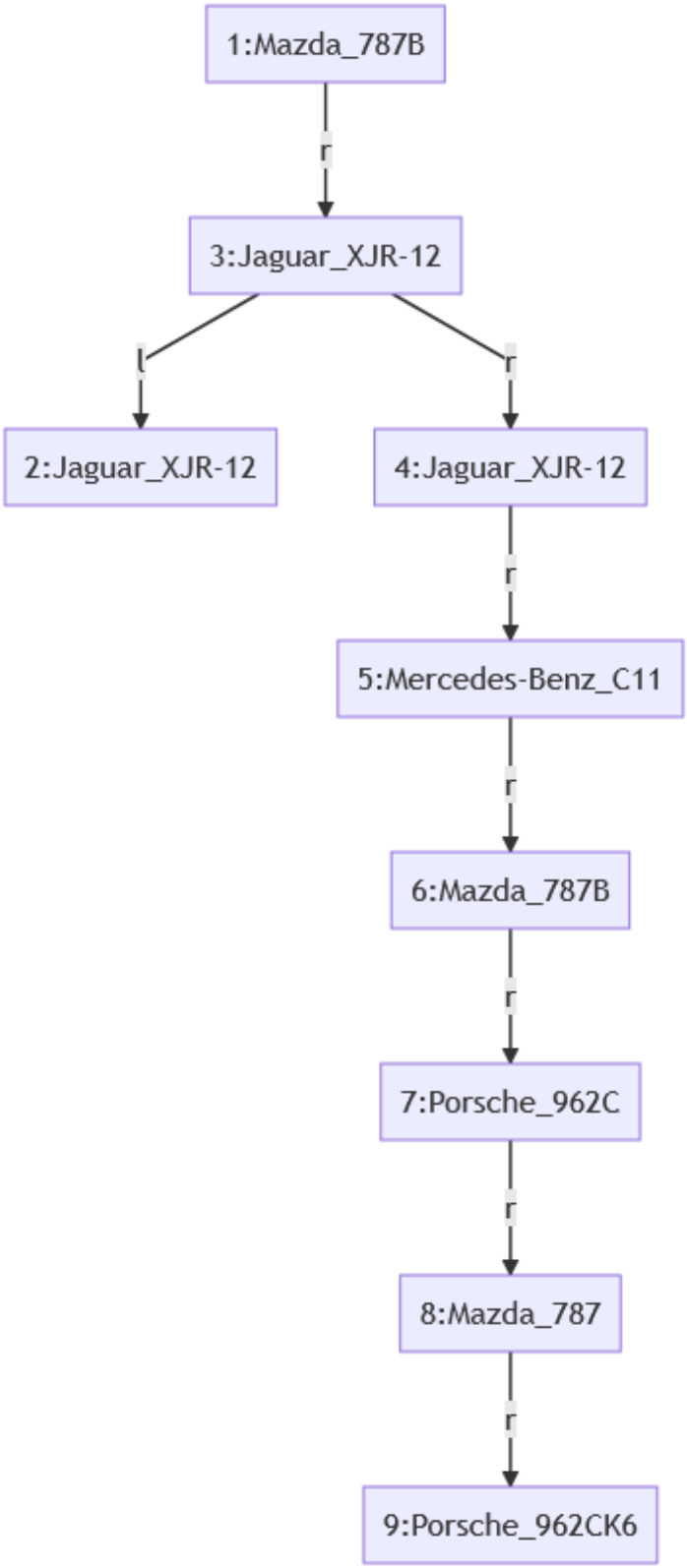


Rappresentazione visiva dell'output di `btree::print()` nel test sugli interi

Successivamente, vengono eseguite le operazioni di stampa (`btree::print()`), di determinazione dell'esistenza di un nodo (`btree::exists()`) e del calcolo delle dimensioni dell'albero (`btree::get_size()`). L'albero viene poi copiato in `bst_b` tramite il costruttore di copia (`btree::btree(const btree&)`), il quale viene, a sua volta, copiato in `bst_c` tramite assegnamento (`btree::operator=()`). Viene poi creato `bst_sub` a partire dal nodo contenente il valore 3 di `bst_a` tramite `btree::subtree()`, e vengono stampati i nodi della coda associata a `bst_a` tramite iteratore (`btree::const_iterator`). Vengono poi determinati e stampati tramite `printIF()` i nodi di valore pari nella coda associata all'albero `bst_a`.

Il test sugli indici etichettati

Del tutto analogo al test sugli interi, questo viene eseguito per dimostrare il funzionamento della classe `btree` indipendentemente dal tipo usato per costruirla. Sono stati quindi definiti la semplice classe `labeled_idx`, che rappresenta un indice di tipo `uint`, avente un etichetta di tipo `std::string` ed i funtori associati. E' stato poi creato un albero come in figura:



Rappresentazione visiva dell'output di `btree::print()` nel test sugli indici etichettati

Che rappresenta (malamente) le prime nove autovetture classificate della 24 ore di Le Mans del 1991.

> ***NOTA***
> Questo test è stato determinante nella scoperta di alcuni problemi nel codice

---

File `Makefile`

All'interno del *Makefile* sono definiti i seguenti target:

1. `debug` - compila il progetto e produce un file eseguibile che contiene i simboli di debug (usando le flag `-Og -ggdb`), rendendolo adatto all'analisi dinamica tramite GDB e Valgrind.
2. `release` - compila il progetto in maniera ottimizzata (`-O3`) e rimuove seguentemente i prodotti intermedi di compilazione invocando `autoclean`
3. `clean` - rimuove tutti i prodotti intermedi di compilazione, escluso il file eseguibile
4. `autoclean` - rimuove i prodotti intermedi di compilazione solo per la build di `release`, invocato automaticamente
5. `check` - esegue un controllo dei memory leak invocando Valgrind con la flag `--leak-check=full` sul file eseguibile

Esso definisce inoltre una variabile contenente, tra l'altro, le flag `-Wall`, `-Wextra` e `-pedantic` per far sì che ogni potenziale errore venga segnalato dalla suite GNU in fase di compilazione.

Non è possibile utilizzare la flag `-std=c++03` in quanto troppo restrittiva: è quindi necessario ripiegare su `-std=c++0x`.

Note stilistiche e di sviluppo

Il codice è stato scritto ed indentato secondo la Google C++ Style Guide, quindi riformattato automaticamente.

L'intero progetto è stato sviluppato sul sistema operativo Debian GNU/Linux 10 "buster", sul kernel Linux 4.19.0-8-amd64. Sono stati utilizzati gli strumenti di sviluppo GNU gcc 8.3.0 e GNU Make 4.2.1 per la compilazione, Valgrind 3.14.0 per l'analisi dinamica del codice e clang e cpplint per l'indentazione automatica e l'analisi statica.

E' inoltre stato impiegato Git come strumento di code versioning, nello specifico avvalendosi di una repository privata su GitHub.

Le funzioni di stampa (nello specifico `btree::print()` e

`btree::operator<<`) stampano l'albero binario di ricerca sotto forma di testo utilizzando una sintassi compatibile con Mermaid, un linguaggio di markup in grado di rappresentare, tra l'altro, grafi e alberi. Nello specifico, i dati inseriti nell'albero sono rappresentati sotto forma di grafo orientato, i cui nodi contengono i dati presenti nel grafo e gli archi sono etichettati in base alla direzione del ramo (`l` - ramo sinistro, `r` - ramo destro). Questo prevede che i dati contenuti nei nodi siano stampabili tramite operatore di stream (in `btree::node::operator<<`) e che inseriscano nello stream una sequenza di caratteri **non contenente** spazi. E' possibile visualizzare i grafi ottenuti tramite Mermaid Live Editor, oppure tramite estensioni di strumenti quali *Pandoc* o *Visual Studio Code*.