

UNIVERSITÀ DEGLI STUDI DI NAPOLI “PARTHENOPE”
DIPARTIMENTO DI SCIENZE E TECNOLOGIE
CORSO DI LAUREA TRIENNALE IN INFORMATICA



ALGORITMI E STRUTTURE DATI E LABORATORIO DI ALGORITMI
E STRUTTURE DATI

Relazione progetto traccia 2

DOCENTI
Prof. Alessio Ferone
Prof. Francesco Camastra

CANDIDATO
Vittorio Fones 0124/1384

Anno Accademico 2019-2020

Indice

I	1
1 Albero red-black di hash table	2
1.1 Descrizione problema	2
1.2 Descrizione strutture dati	2
1.2.1 Alberi binari di ricerca	2
1.2.2 Alberi Red-Black	3
1.2.3 Hash Table ad indirizzamento aperto	3
1.3 Formato di input e di output	4
1.3.1 Input	4
1.3.2 Output	4
1.4 Descrizione algoritmo	5
1.4.1 Pseudo codice	5
1.4.2 Diagrammi delle classe e dettagli architetturali	6
1.4.3 Hash Table	8
1.4.4 Red Black Hash	10
1.5 Studio complessità	11
1.6 Test e risultati	12
1.6.1 Test effettuati	12
1.7 Codice sorgente	14
 II	 43
2 Viaggi Galattici	44
2.1 Descrizione problema	44
2.2 Descrizione strutture dati	44
2.2.1 Grafi	44
2.2.2 Coda di priorità	45
2.2.3 Min Heap Binario	45
2.3 Formato di input e di output	46

2.3.1	Input	46
2.3.2	Output	46
2.4	Descrizione algoritmo	47
2.4.1	Pseudo codice	47
2.4.2	Diagrammi delle classe e dettagli architetturali	48
2.5	Studio complessità	53
2.6	Test e risultati	54
2.6.1	Test effettuati	54
2.7	Codice sorgente	59

Elenco delle figure

1.1	Nodi degli alberi	7
1.2	Alberi	8
1.3	Hash Table	9
1.4	Parser e RBH	10
2.1	Min priority Queue	49
2.2	Vertici e Grafo	51
2.3	Loader e Grago galattico	52
2.4	Grafo non connesso.	54
2.5	Grafo a diamante.	55
2.6	Grafo d'esempio.	56

Indice dei codici sorgente

1.1	debug.hpp	14
1.2	item.hpp	14
1.3	genericnode.hpp	15
1.4	abstractbinarynode.hpp	16
1.5	color.hpp	17
1.6	redblacknode.hpp	18
1.7	hashtable.hpp	19
1.8	hashtable.cpp	21
1.9	binarysearchtree.hpp	24
1.10	binarysearchtree.cpp	25
1.11	redblacktree.hpp	29
1.12	redblacktree.cpp	30
1.13	redblackhash.hpp	35
1.14	redblackhash.cpp	36
1.15	parser.hpp	38
1.16	parser.cpp	39
2.17	heap.hpp	59
2.18	heap.cpp	61
2.19	minHeap.hpp	62
2.20	minHeap.cpp	63
2.21	minPriorityQueue.hpp	64
2.22	minPriorityQueue.cpp	65
2.23	debug.hpp	66
2.24	item.hpp	67
2.25	vertex.hpp	67
2.26	vertex.cpp	69
2.27	graph.hpp	69
2.28	graph.cpp	70
2.29	galacticgraph.hpp	75
2.30	galacticgraph.cpp	75

2.31 loader.hpp	77
2.32 loader.cpp	77

Parte I

Capitolo 1

Albero red-black di hash table

1.1 Descrizione problema

Il problema in analisi prevede di creare una struttura dati, in C++, che d'ora in avanti chiameremo **red-black hash**, in grado di immagazzinare delle stringhe alfanumeriche. Tale struttura è l'unione di un **albero rosso-nero**, o **albero red-black**, (albero binario di ricerca autobilanciato), e delle **tavole di hash** (array associativi usati creare mappe tra chiavi e valori): all'interno di ogni nodo di tale albero, vi è presente una **hash table**, struttura dati che associa per ogni chiave un singolo valore, al cui interno sono presenti delle stringhe. La traccia prevedeva di poter effettuare operazioni **C.R.D.**¹ su tuple nel formato **chiave1:chiave2:stringa**. La chiave 1 indicizza un nodo dell'albero red black, mentre la chiave 2 viene usata per associare la stringa all'interno della hashtable. Vi è quindi una relazione 1:1 per i nodi dell'albero e l'hash table, e 1:M tra l'hash table e le stringhe, dove **M** è la dimensione massima dell'hash table.

1.2 Descrizione strutture dati

1.2.1 Alberi binari di ricerca

Gli **alberi binari di ricerca** sono delle strutture dati che immagazzinano dati in un albero avente in ogni nodo due figli. Gli **ABR** (o in inglese BST) godono della seguente proprietà: $\forall x \in \text{BST}: \text{key}(x.\text{left}) \leq \text{key}(x) < \text{key}(x.\text{right})$. Ovvero ogni nodo in un ABR ha come valore della chiave un valore maggiore del figlio sinistro ma minore di quello destro.

Ciò assicura operazioni in una complessità **logaritmica** data dalla profondità dell'al-

¹Create Retrieve Delete. Operazioni tipiche delle basi di dati, ma senza la possibilità di effettuare Updates.

bero. Il problema sorge nel caso in cui avvengono frequenti cancellazioni che seppur mantengono la proprietà degli ABR, degradano tale albero in una lista concatenata portando le operazioni di inserimento, cancellazione e ricerca ad avere complessità lineare.

1.2.2 Alberi Red-Black

Gli alberi red black sono degli alberi binari di ricerca **autobilancianti**. Ogni volta che si inserisce un nuovo nodo, o lo si cancella si effettuano delle operazioni per bilanciare l'albero. Gli alberi rosso neri posseggono **5 proprietà** utili ai metodi di supporto *insertFix()* e *deleteFix()* per garantire che le complessità peggiori abbiano al più come valore l'altezza dell'albero, ovvero $O(\log_2 n)$. I metodi di supporto all'inserimento e alla cancellazione fanno uso delle rotazioni di un nodo, operazione che permette di compattare l'albero e garantirne il corretto bilanciamento. Di seguito verranno elencate tali proprietà:

- ogni nodo è rosso o nero;
- la radice è nera;
- ogni foglia è nera;
- se un nodo è rosso, allora entrambi i suoi figli sono neri;
- per ogni nodo, tutti i cammini semplici che vanno dal nodo alle foglie sue discendenti contengono lo stesso numero di nodi neri.

1.2.3 Hash Table ad indirizzamento aperto

Le **Hash Table**, o hash map, sono delle strutture dati che permettono di associare ad una chiave un singolo valore. Quando si vuole memorizzare un dato si usa la sua chiave e vi si applica una **funzione di hashing** che ne calcolerà un indice usato per inserire nell'array tale dato. Può capitare però che una funzione hash applicata su chiavi diverse indicizzi celle simili dell'array, per tanto bisogna gestire queste *collisioni*. La tecnica usata per risolvere le collisioni è del tipo a **indirizzamento aperto**, ovvero non facendo uso dei puntatori si *ispeziona* l'hashtable fino a incontrare una posizione libera, se presente. Il metodo di ispezione scelto è quello del **doppio hashing**: rispetto ad altre ispezioni, quella del doppio hashing trova una posizione in modo più veloce. Nei capitoli successivi vedremo nel dettaglio il funzionamento.

1.3 Formato di input e di output

1.3.1 Input

Il programma lavora su tuple nel formato *chiave1 : chiave2 : stringa* e i dati in input sono dati da:

- file di testo da selezionare all'avvio contenente le tuple;

```
$ ./rbhash inputs/asd.txt
```

- opzione da tastiera per effettuare operazioni sulle tuple.

```
>_ 1:2:stringa
```

Il programma leggerà le righe e allocherà nodi e hashtable in base alle chiavi date in input.

1.3.2 Output

L'output del programma è un menu contestuale in cui l'utente può effettuare le operazioni mostrate di seguito:

Tutte le opzioni restituiscono un output di avvenuta operazione con dettagli su eventuali errori, ad eccezione della stampa (opzione 4) che restituisce in output l'intera struttura dati caricata in memoria.

1.4 Descrizione algoritmo

1.4.1 Pseudo codice

L'**inserimento** nella struttura dati creata va effettuare prima una ricerca del nodo di chiave *key1*. Se dovesse riscontrare un esito negativo si procede alla allocazione di un hashtable e un nuovo nodo. In caso contrario si procede alla ricerca di una cella della tavola di hash con la *key2*, se anche questa ricerca restituisce un esito negativo allora si procede con l'inserimento.

Algorithm: Insert

```

input: key1, key2, string
result: true if successfull, false if not
1 node = Search in Red Black with key1;
2 if  $\nexists$  node :
3   | new Hashtable();
4   | new Node();
5   | Hashtable.insert(key2, string);
6   | Node.insert(key1, Hashtable);
7   | return true;
8 else:
9   | if  $\nexists$  node.Hashtable.search(key2) :
10  |   | node.Hashtable.insert(key2, d);
11  |   | return true;
12 return false;

```

La **ricerca** controlla la correttezza delle chiavi e della stringa inserita nella tupla: in caso di riscontro positivo la function ritornerà il nodo dell'albero. Questo comportamento è stato previsto per poter evitare di effettuare una o più ricerche nella procedura di cancellazione.

Algorithm: Search (retrieve)

```

input: key1, key2, string
result: node if found or NIL if not found
1 node = Search in Red Black with key1;
2 if  $\exists$  node :
3   | data = node.Hashtable.search(key2);
4   | if string = data :
5   |   | return node;
6 return NIL;

```

La **cancellazione** di una tupla effettua una operazione di ricerca nella struttura dati. Se la ricerca ha riscontro positivo allora si procede con i due casi della rimozione: se la chiave secondaria indicizza una hashtable in cui vi è presente un solo elemento, allora si cancellerà l'intero nodo red black, altrimenti si procede alla normale rimozione dalla hashtable.

Algorithm: Remove (delete)

```

input : key1, key2, string
result: true if deleted or false if not
1 node = Search in Red Black Hash;
2 if node  $\neq$  NIL :
3   if node.Hashtable.capacity = 1 :
4     delete node;
5   else:
6     node.Hashtable.remove(key2);
7   return true;
8 return false;
```

1.4.2 Diagrammi delle classe e dettagli architetturali

Nodi Binari

I nodi facenti parti degli alberi binari sono nodi che derivano da una classe astratta che a sua volta estende un nodo generico.

Per permettere agli alberi binari di usare come parametro template un generico nodo, si è sfruttato un particolare pattern strutturale chiamato **CRTP** (*Curiously recurring template pattern*): una classe (e.g. `ConcreteBinaryNode`) eredita da una classe astratta che usa il CRTP (e.g. `AbstractBinaryNode` ha come parametro template un generico nodo), usando come parametro di specializzazione se stessa. Il nodo concreto `RedBlack` in più implementa la classe `color`. Si è scelta tale tecnica che non porta miglioramenti sostanziali al codice se non quella di nascondere dettagli implementativi e un' indipendenza dalla rappresentazione in memoria del colore.

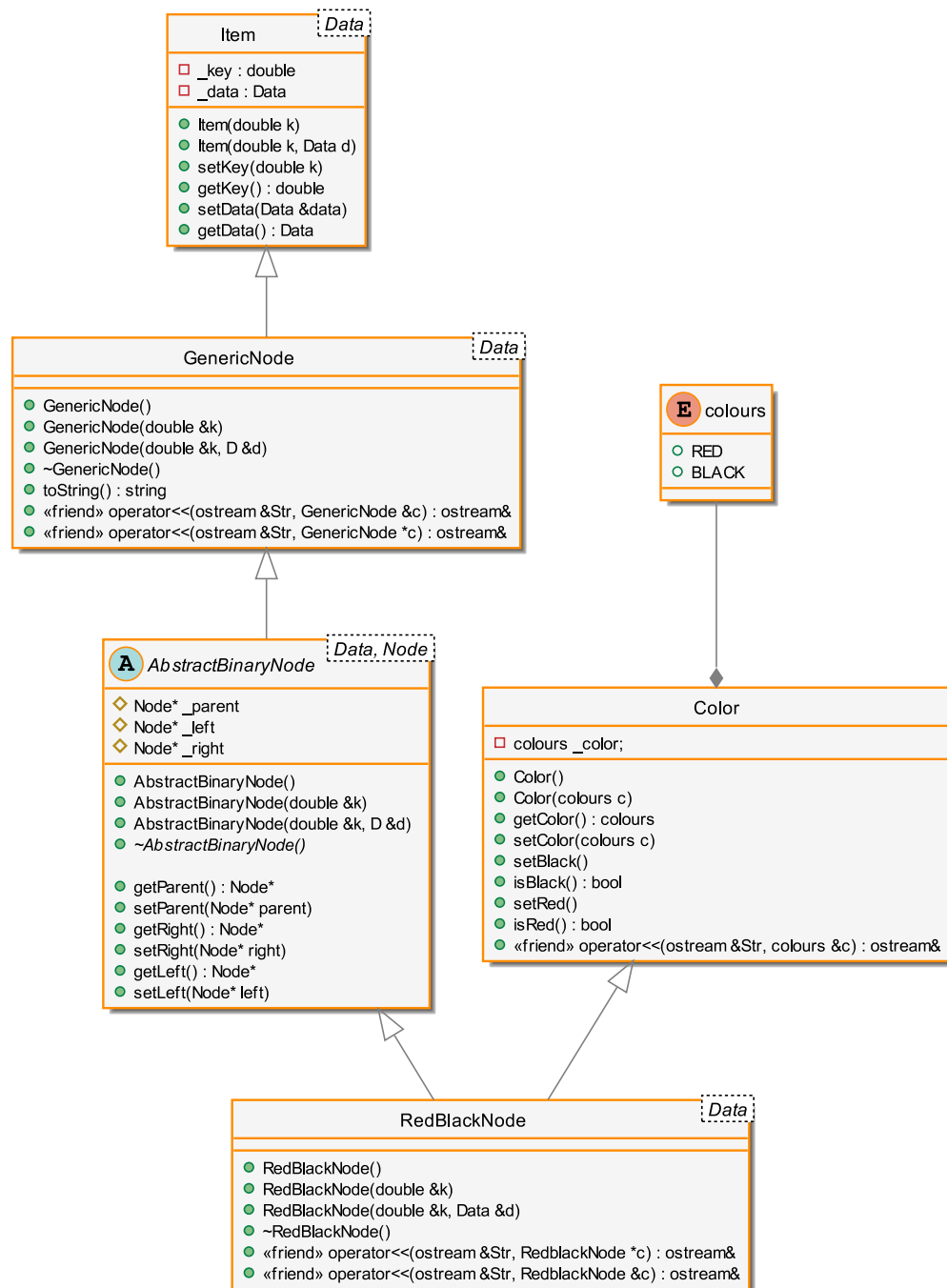


Figura 1.1: Nodi degli alberi

Alberi

Un albero Rosso Nero è un albero binario di ricerca auto bilanciante, per tanto si è scelto di estendere la classe `BinarySearchTree` ed aggiungere i metodi di supporto al bilanciamento. Inoltre due metodi virtuali (`insert` e `delete`), sono stati ridefiniti nell'implementazione del Red Black. `insertNode()` nella class `RedBlackTree` richia-

ma tramite scope la insert di BinarySearchTree e successivamente applica il **fix** dei red black.

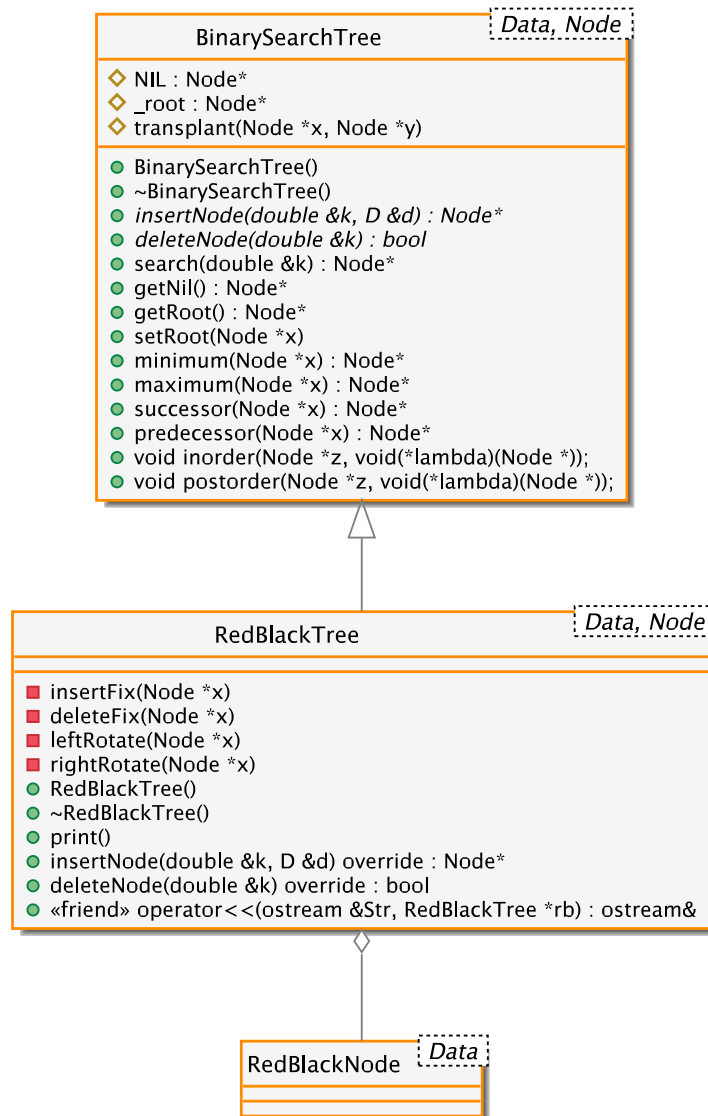


Figura 1.2: Alberi

1.4.3 Hash Table

Onde evitare di dover riscrivere codice si è scelto di sfruttare il nodo generico anche per l'**HashTable**. A tal proposito si è sviluppato una tabella hash ad **indirizzamento aperto** che prende in input come paramentro templatato, il dato da conservare. Per risolvere le collisioni si è scelto di usare due funzioni hash: **k** è un valore double che indica la chiave, **i** è l'iteratore che al massimo **m** volte ² applicherà la funzione di hash. La prima restituisce un indice con valore compreso tra $[0, m]$, mentre la

²m è il size dell'hashtable

seconda funzione di hash un valore compreso tra $[1, m - 1]$. Verrà usata la prima funzione di hash e in caso di collisione si riapplicherà $h(i, m) = (h_1(k) + i * h_2(k)) \bmod m$. Sebbene con un costo computazionale più alto, il doppio hashing risolve le collisioni più in fretta rispetto all'ispezione lineare o quadratica.

La funzione `search()` nella class **HashTable** è stata sovraccaricata per restituire valori diversi. Con la ricerca per valore si restituisce, se disponibile un indice che usato nella seconda funzione di ricerca restituisce il dato.

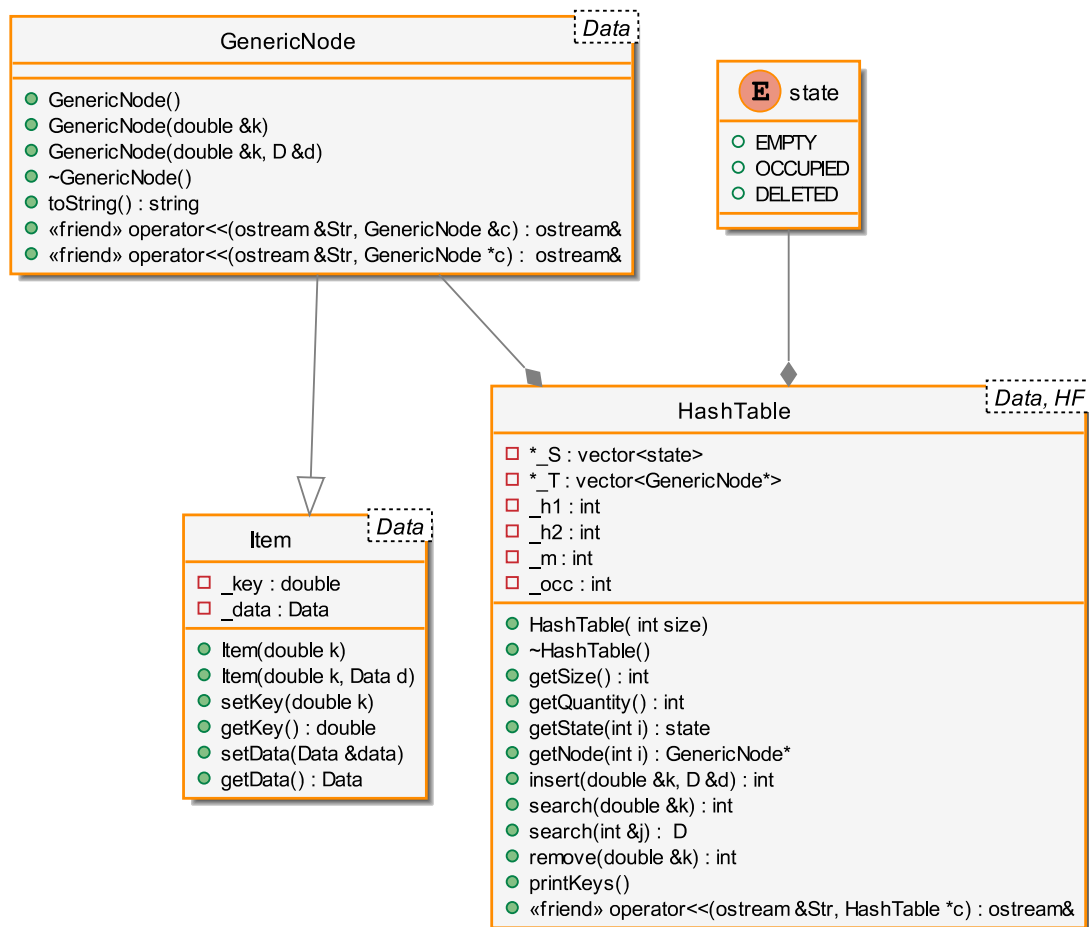


Figura 1.3: Hash Table

1.4.4 Red Black Hash

La struttura dati realizzata fa uso degli alberi red black e delle tavole di hash come spiegato nel primo paragrafo. Si è scelto di usare una classe **Parser** per riempire la struttura dati delle tuple contenute nel file di testo o di quelle inserite tramite riga di comando.

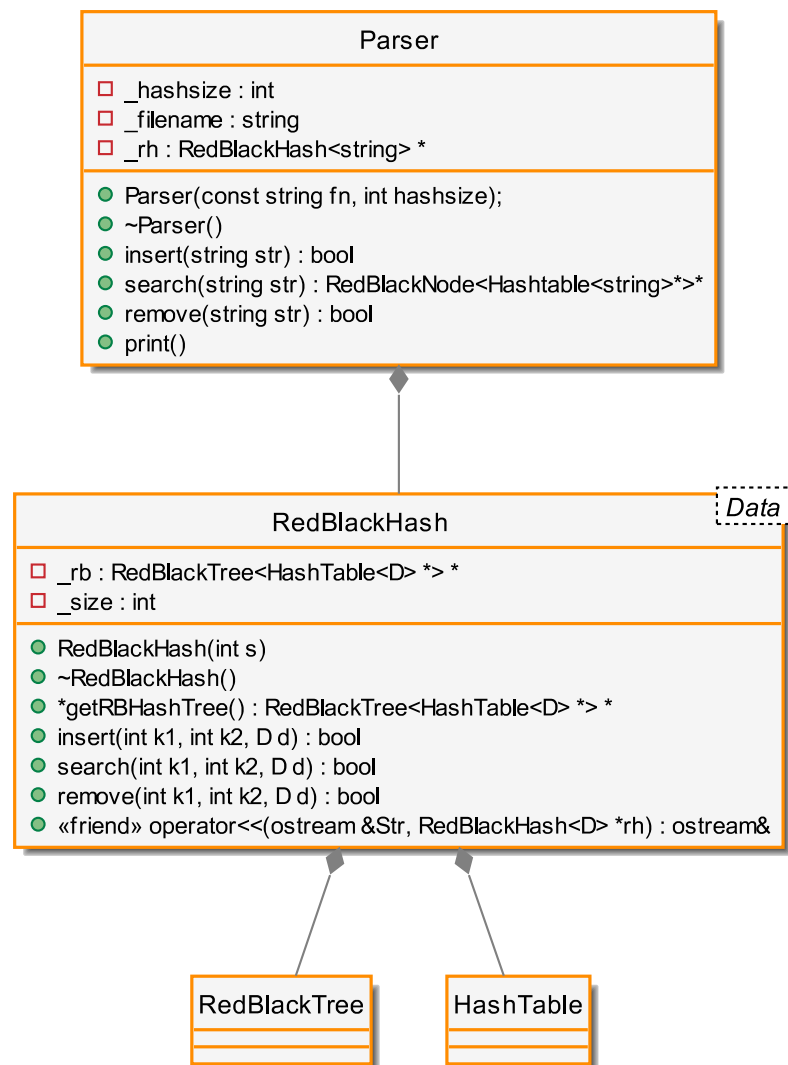


Figura 1.4: Parser e RBH

1.5 Studio complessità

Come detto precedentemente gli alberi Red Black hanno una complessità temporale nel **caso peggiore** al più equivalente ad $O(\log_2 n)$ poiché bisogna scorrere tutto l'albero in profondità.

Le tabelle di hash a indirizzamento aperto, con doppia funzione di hash hanno complessità temporali prossime all'*hashing* 'ideale'. Per poter assicurare che le due funzioni di hash producano complessità nel caso medio uguali a $O(1)$, si deve scegliere un valore della tavola di hash uguale ad un numero primo o come potenza di due, e usare la seconda funzione di hash (quella che scandisce l'offset dato dalle ispezioni successive) con un valore poco più piccolo di m ($m-1$ o $m-2$). In tal modo il doppio hashing usa $O(m^2)$ sequenze di ispezione, perché ogni coppia $(h_1(k), h_2(k))$ produce una distinta sequenza di ispezione. Le **hashtable nel caso migliore**, ovvero non avendo nessuna collisione inseriscono, cercano e cancellano i dati in $O(1)$. Nel caso **peggiore** avremo un tempo massimo di $O(m)$, dovuto alla scansione lineare di tutta la tavola di hash: ciò è dovuto alla hashtable che si riempie cioè quando il **fattore di carico** (rapporto di elementi inseriti e size della tavola) α tende a 1.

Nella struttura dati **red-black hash** quando effettuiamo un **inserimento** andremo prima ad effettuare una ricerca nell'albero rosso nero ($O(\log_2 n)$):

- se esiste il nodo di chiave 1 allora effettuiamo una ricerca nell'hash table tramite la chiave 2, e se possibile concludiamo l'inserimento. Nel caso medio avremo una complessità del tipo $O(\log_2 n) + O(1) + O(1)$ e $O(\log_2 n) + O(m) + O(m)$ se l'hashtable ha un fattore di carico prossimo all'1. In altri termini **$O(\log_2 n)$** e **$O(m + \log_2 n)$** ;
- se invece il nodo di chiave 1 non esiste, dobbiamo allocare una hashtable, mappare la stringa alla chiave 2 e inserirla nell'albero: ciò porta una complessità pari a $O(\log_2 n) + O(1) + O(\log_2 n)$ ovvero **$O(\log_2 n)$** .

L'operazione di **ricerca** impiega **$O(m + \log_2 n)$** nel caso in cui esista il nodo con chiave 1 e la ricerca nella tavola hash restituisca l'indice in un tempo lineare; l'uso di tale indice serve per accedere al dato in $O(1)$. Nel caso migliore invece la ricerca avviene in **$O(\log_2 n)$** .

La **cancellazione** ha complessità uguali alle precedenti: si effettua una ricerca e se l'hashtable ha solo un elemento allora si provvede a cancellare l'intero nodo dall'albero, altrimenti si cancella solamente nell'hashtable: pertanto **$O(\log_2 n)$** oppure **$O(m + \log_2 n)$** .

Tramite questo prospetto possiamo affermare che i casi peggiori sono dettati dalla tavola di hash e dipende tutto dal suo fattore di carico.

1.6 Test e risultati

1.6.1 Test effettuati

Nei test effettuati sono stati usati due file e tramite funzioni di debug per la stampa si è cercato di verificare per quanto possibile l'inserimento e cancellazione dell'albero red black. Di seguito verranno mostrati alcuni esempi estrapolati dalla shell.

Notare come nell'output di stampa della struttura dati non vi sono presenti le chiavi duplicate. Inoltre si è scelto di stampare tramite una **visita inorder** per avere un ordine crescente delle chiavi dell'albero.

```
$ ./rbhash inputs/asd
```

```
**** MENU ****
```

```
1. Insert
```

```
2. Remove
```

```
3. Query
```

```
4. Print
```

```
0. Exit
```

```
>_ 4
```

```
'(1) :
```

```
    [4] : test'
```

```
(2) :
```

```
    [4] : 54
```

```
    [12] : asd
```

```
'(123) :  
      [321] : hello'
```

L'eliminazione della radice, ovvero il nodo nero 2 comporta un bilanciamento con conseguente ricolorazione di nodi.

```
*****WARNING*****
```

```
Insert tuple (key1:key2:data) to delete -> 2:4:54
```

```
*****WARNING*****
```

```
Insert tuple (key1:key2:data) to delete -> 2:12:asd
```

```
'(1) :  
      [4] : test'  
(123) :  
      [321] : hello
```

1.7 Codice sorgente

Codice 1.1: debug.hpp

```

1  #ifndef _DEBUG_HPP_
2  #define _DEBUG_HPP_
3
4  #include <iostream>
5
6  /**
7   * printer and debugging purpose
8   */
9
10 #define debug_print std::cerr
11
12 #ifdef DEBUG
13     #define IFDEBUG if(1)
14 #else
15     #define IFDEBUG if(0)
16 #endif
17
18 #endif //_DEBUG_HPP_

```

Codice 1.2: item.hpp

```

1  #ifndef _ITEM_HPP_
2  #define _ITEM_HPP_
3
4  #include <limits>
5
6  /**
7   * @brief Generic Item
8   * @details using a double as a key
9   * @tparam Data parameter as value
10 */
11 template <class Data>
12 class Item {
13     private:
14         double _key;
15         Data _data;
16
17     public:
18         Item() : _key(std::numeric_limits<double>::min()), _data {} {}
19         Item(double &k) : _key(k), _data {} {}
20         Item(double &k, Data &d) : _key(k), _data(d) {}
21

```

```

22     //generic setter and getter
23     void setKey(double &k) { this->_key = k; }
24     double& getKey() { return this->_key; }
25
26     void setData(Data &data) { this->_data = data; }
27     Data& getData() { return this->_data; }
28 };
29
30 #endif // _ITEM_HPP_

```

Codice 1.3: genericnode.hpp

```

1  #ifndef _GENERICNODE_HPP_
2  #define _GENERICNODE_HPP_
3
4  #include <string>
5  #include <iostream>
6
7  #include <other/item.hpp>
8
9  /**
10   * @brief Generic Node class
11   * @tparam N template base class
12   */
13  template <typename D>
14  class GenericNode : public Item<D> {
15  public:
16      GenericNode() : Item<D>() {};
17      GenericNode(double &k) : Item<D>(k) {};
18      GenericNode(double &k, D &d) : Item<D>(k, d) {};
19      ~GenericNode() {};
20      std::string toString(){return
21          ↪ "["+std::to_string((int)this->getKey())+"] : "+
22          std::string(this->getData())+"\n";}
23
24      friend std::ostream& operator<<(std::ostream &Str, GenericNode<D> &c){
25          Str<<c.toString(); return Str; }
26
27      friend std::ostream& operator<<(std::ostream &Str, GenericNode<D> *c){
28          Str<<c->toString(); return Str; }
29  };
30  #endif // _GENERICNODE_HPP_

```

Codice 1.4: abstractbinarynode.hpp

```

1  #ifndef _IBNODE_HPP_
2  #define _IBNODE_HPP_
3
4  #include <iostream>
5
6  #include <nodes/genericnode.hpp>
7
8  /**
9    * @brief AbstractBinaryNode
10   * @details Using CRTP as base (Used in RNode for example)
11   * https://en.wikipedia.org/wiki/Curiously\_recurring\_template\_pattern
12   * @tparam D data template for value to store in node
13   * @tparam N template for CRTP pattern
14   */
15  template <typename D, class Node>
16  class AbstractBinaryNode : public GenericNode<D> {
17  protected:
18      // since is a Binary Node we got a parent and only 2 child
19      Node* _parent = nullptr; // defaults values
20      Node* _left = nullptr;
21      Node* _right = nullptr;
22
23  public:
24      AbstractBinaryNode() : GenericNode<D>() {};
25      AbstractBinaryNode(double &k) : GenericNode<D>(k) {};
26      AbstractBinaryNode(double &k, D &d) : GenericNode<D>(k, d) {};
27      virtual ~AbstractBinaryNode() {};
28
29      //various setter and getter
30      Node* getParent(){ return this->_parent; };
31      void setParent(Node* parent){ this->_parent=parent; };
32
33      Node* getRight(){ return this->_right; };
34      void setRight(Node* right){ this->_right=right; };
35
36      Node* getLeft(){ return this->_left; };
37      void setLeft(Node* left){ this->_left=left; };
38  };
39
40 #endif //_IAbstractBNode_HPP_

```

Codice 1.5: color.hpp

```
1  #ifndef _COLOR_HPP_
2  #define _COLOR_HPP_
3
4  #include <iostream>
5
6  /**
7   * @brief Colours tag as enum
8   */
9  enum colours {RED, BLACK};
10
11
12 /**
13  * @brief Color setter of a node
14  * @details used for generic implementation
15  */
16 class Color {
17     private:
18         colours _color;
19
20     public:
21         Color();
22         Color(colours c) : _color(c) {};
23
24         //various setter and getter
25         colours getColor(){return this->_color;}
26         void setColor(colours c){this->_color = c;}
27
28         void setBlack(){this->_color = BLACK;}
29         bool isBlack(){return (this->_color == BLACK);}
30
31         void setRed(){this->_color = RED;}
32         bool isRed(){return (this->_color == RED);}
33
34         friend std::ostream& operator<<(std::ostream &Str, colours &c){
35             (c==RED) ? Str<<"red" : Str<<"black";
36             return Str;
37         }
38 };
39
40 #endif //_COLOR_HPP_
```

Codice 1.6: redblacknode.hpp

```

1  #ifndef _REDBLACKNODE_HPP_
2  #define _REDBLACKNODE_HPP_
3
4  #include <nodes/abstractbinarynode.hpp>
5  #include <nodes/color.hpp>
6
7  /**
8   * @brief Red Black Node
9   * @details Inherits from Binary Node and using
10   * CRTP for a dynamic inheritance. Also extending Color
11   * @tparam D
12   */
13 template <typename D>
14 class RedBlackNode : public AbstractBinaryNode<D, RedBlackNode<D>>, public
    ↳ Color {
15     public:
16         RedBlackNode() : Color(BLACK) {};
17         RedBlackNode(double &k) : AbstractBinaryNode<D, RedBlackNode<D>>(k),
            ↳ Color(BLACK) {};
18         RedBlackNode(double &k, D &d) : AbstractBinaryNode<D,
            ↳ RedBlackNode<D>>(k, d), Color(BLACK) {};
19         ~RedBlackNode() {
20             if(this->_left==nullptr)
21                 delete this->_left;
22             if(this->_right==nullptr)
23                 delete this->_right;
24             if(this->_parent==nullptr)
25                 delete this->_parent;
26         };
27
28         friend std::ostream& operator<<(std::ostream &Str, RedBlackNode<D> &c)
            ↳ {
29             if(c.isRed())
30                 Str<<"\033[31m";
31             Str<<"("<<c.getKey()<<" ) : \n"<<c.getData()<<"\033[0m";
32             return Str;
33         }
34
35         friend std::ostream& operator<<(std::ostream &Str, RedBlackNode<D> *c)
            ↳ {
36             if(c->isRed())
37                 Str<<"\033[31m";
38             Str<<"("<<c->getKey()<<" ) : \n"<<c->getData()<<"\033[0m";
39             return Str;
40         }

```



```

41 };
42
43
44 #endif // _REDBLACKNODE_HPP_

```

Codice 1.7: hashtable.hpp

```

1  #ifndef _HASHTABLE_HPP_
2  #define _HASHTABLE_HPP_
3
4  #include <limits>
5  #include <vector>
6  #include <cmath>
7
8  #include <nodes/genericnode.hpp>
9
10
11  /* Enum for state of single HashNode */
12  enum state { EMPTY, OCCUPIED, DELETED };
13
14  /**
15   * @brief HashTable datastructure
16   * @tparam D is the Data to store
17   * @tparam HashFunction is the class that make the hashfunction
18   * to compute the index of node
19   */
20  template <typename D>
21  class HashTable {
22  private:
23      std::vector<state> *_S; // array of states
24      std::vector<GenericNode<D>*> *_T; // effective hashtable
25
26      // using fmod -> double % int
27      int _h1(double &k){return static_cast<int>(std::fmod(k, this->_m));}
28      int _h2(double &k){return 1 + static_cast<int>(std::fmod(k,
29          ↪ this->_m-1));}
30
31      int _m; // default capacity
32      int _occ = 0; // number of occupied nodes
33
34  public:
35      HashTable( int size = 701 );
36      ~HashTable();
37
38      /**
39       * @brief Get the capacity of hashtable
40       * @return int

```

```

40     */
41     int getSize();
42
43     /**
44      * @brief Get the current hashnode in hashtable
45      * @return int
46      */
47     int getQuantity();
48
49     /**
50      * @brief Get the state of single node
51      * @param i
52      * @return state
53      */
54     state getState(int i);
55
56     /**
57      * @brief Get the Node object
58      * @param i
59      * @return HashNode<D>*
60      */
61     GenericNode<D>* getNode(int i);
62
63
64     /**
65      * @brief insert a data in HashTable
66      * @param k is the key
67      * @param d is the value
68      * @return int is the index, return -1 if there isn't space left
69      */
70     int insert(double &k, D &d);
71
72     /**
73      * @brief search a key in HashTable
74      * @param k is the key
75      * @return int is the index, return -1 if there isn't
76      */
77     int search(double &k);
78
79     /**
80      * @brief Overload of search method
81      * @param j is the index
82      * @return D is the value returned
83      */
84     D search(int &j);
85
86     /**
87      * @brief remove a Node with that key
88      * @param k is the key

```

```

89     * @return int is the index
90     */
91     int remove(double &k);
92
93     /**
94     * @brief print all node's keys;
95     */
96     void printKeys();
97
98     /**
99     * @brief Override of operator<< for print
100    * @param Str ostream obj
101    * @param c HashTable<D>
102    * @return std::ostream&
103    */
104    friend std::ostream& operator<<(std::ostream &Str, HashTable<D> *c) {
105        // put every HashNode<D> that there isn't EMPTY or DELETED
106        for(auto i = 0; i < c->getSize(); i++)
107            if(c->getState(i) == OCCUPIED)
108                Str<<"\t"<<c->getNode(i);
109        Str<<"\n\n";
110        return Str;
111    };
112 };
113
114
115
116 #endif //_HASHTABLE_HPP_

```

Codice 1.8: hashtable.cpp

```

1  #include <iostream> // cerr
2
3  #include <hashables/hashtable.hpp>
4
5
6  //creating GenericNode vector and state vector
7  template <typename D>
8  HashTable<D>::HashTable(int size) : _m(size) {
9      this->_T = new std::vector<GenericNode<D>*>(this->_m, nullptr);
10     this->_S = new std::vector<state>(this->_m, EMPTY); //init with EMPTY
11     ↪ state
12 }
13
14 //destructing hashnodes
15 template <typename D>

```

```

16 HashTable<D>::~~HashTable() {
17     for(auto t = 0; t != this->_m; t++)
18         if(this->_S->at(t) != EMPTY)
19             delete this->_T->at(t);
20
21     delete []_T;
22     delete []_S;
23 }
24
25
26 //getter hash table size
27 template <typename D>
28 int HashTable<D>::getSize() {
29     return this->_m;
30 }
31
32
33 //return the state of a indexed (j) GenericNode
34 template <typename D>
35 state HashTable<D>::getState(int i) {
36     if(i >= 0 && i < this->_m)
37         return this->_S->at(i);
38     return DELETED;
39 }
40
41
42 // return an GenericNode indexed by i
43 template <typename D>
44 GenericNode<D>* HashTable<D>::getNode(int i) {
45     if(i >= 0 && i < this->_m)
46         return this->_T->at(i); // return nullptr if is not
47         //allocated
48     return nullptr; // return this null if there i is not valid
49 }
50
51 // open addressable hashtable
52 template <typename D>
53 int HashTable<D>::insert(double &k, D &d) {
54     int i = 0;
55     int index = this->_h1(k);
56     int j = index;
57     while(i != this->_m) {
58         if(this->_S->at(j) != OCCUPIED) {
59             this->_T->at(j) = new GenericNode<D>(k, d);
60             this->_S->at(j) = OCCUPIED; // set STATE
61             this->_occ++;
62             return j;
63         }
64         else

```

```

65         j = ( index + (i++) * this->_h2(k)) % this->_m; // double
           ↪ HashFunction
66     }
67
68     std::cerr<<"HashTable: overflow."<<std::endl;
69     return -1; // no space left
70 }
71
72
73 // searching in a open addressable hash table
74 template <typename D>
75 int HashTable<D>::search(double &k) {
76     int i = 0;
77     int index = this->_h1(k);
78     int j = index;
79     while(i != this->_m) {
80         if(this->_S->at(j) == OCCUPIED && this->_T->at(j)->getKey() == k)
81             return j;
82         else
83             j = ( index + (i++) * this->_h2(k)) % this->_m; // double
           ↪ HashFunction
84     }
85     return -1; // there is no Hash Node with that key
86 }
87
88
89 // return data of Hash Node
90 template <typename D>
91 D HashTable<D>::search(int &j) {
92     if(this->_S->at(j) == OCCUPIED)
93         return this->_T->at(j)->getData();
94     else
95         return nullptr;
96 }
97
98
99 // remove an Hash Node
100 template <typename D>
101 int HashTable<D>::remove(double &k) {
102     int j = this->search(k);
103     if(j > -1 && this->_S->at(j) != DELETED) {
104         this->_S->at(j) = DELETED;
105         delete this->_T->at(j);
106         this->_occ--;
107     }
108     return j;
109 }
110
111

```

```

112 template <typename D>
113 void HashTable<D>::printKeys() {
114     for(auto i = 0; i < this->_m; i++)
115         if(this->_S->at(i) == OCCUPIED)
116             std::cout<<this->_T->at(i)->getKey()<<" ";
117     std::cout<<"\n\n";
118 }
119
120 // return the current quantity of hashnodes
121 template <typename D>
122 int HashTable<D>::getQuantity() {
123     return this->_occ;
124 }

```

Codice 1.9: binarysearchtree.hpp

```

1  #ifndef _BINARYSEARCHTREE_HPP_
2  #define _BINARYSEARCHTREE_HPP_
3
4
5  /**
6   * @brief Binary Search Tree
7   * @details BST inherits from a Generic tree.
8   * ndr. must use a Binary Node as template argument
9   * @tparam D data to store in nodes
10  * @tparam N node argument
11  */
12 template <typename D, class Node>
13 class BinarySearchTree {
14     protected:
15         //generic nil node for space optimization
16         Node *NIL = nullptr;
17         Node *_root = nullptr;
18
19         //transplant a node and link father
20         void transplant(Node *x, Node *y);
21
22     public:
23         BinarySearchTree();
24         ~BinarySearchTree();
25
26         // inserting
27         virtual Node* insertNode(double &k, D &d);
28
29         // deleting
30         virtual bool deleteNode(double &k);
31

```

```

32     //search of a node with a key
33     Node* search(double &k);
34
35     //setter getter for NIL
36     Node* getNil() { return this->NIL; }
37     Node* getRoot() { return this->_root; }
38     void setRoot(Node *x) { this->_root = x; }
39
40     // minimum in a subtree
41     Node* minimum(Node *x);
42
43     //maximum of a subtree
44     Node* maximum(Node *x);
45
46     //successor of a node
47     Node* successor(Node *x);
48
49     //predecessor of a node
50     Node* predecessor(Node *x);
51
52
53     // visit (using pointer function -> c++11 lambda)
54     void inorder(Node *z, void(*lambda)(Node *));
55     //mainly use postoder for distructor
56     void postorder(Node *z, void(*lambda)(Node *));
57
58 };
59
60 #endif //_BINARYSEARCHTREE_HPP_

```

Codice 1.10: binarysearchtree.cpp

```

1  #include <trees/binarysearchtree.hpp>
2  #include <other/debug.hpp>
3
4  // binary search tree constructor
5  template<typename D, class Node>
6  BinarySearchTree<D, Node>::BinarySearchTree() {
7      //allocating NIL and pointing root to NIL
8      this->NIL = new Node();
9
10     this->NIL->setParent(this->NIL);
11     this->NIL->setLeft(this->NIL);
12     this->NIL->setRight(this->NIL);
13
14     this->_root = this->NIL;
15     this->_root->setParent(this->NIL);

```

```

16     this->_root->setLeft(this->NIL);
17     this->_root->setRight(this->NIL);
18 }
19
20 // Binary Search Tree destructor (using post order)
21 template<typename D, class Node>
22 BinarySearchTree<D, Node>::~BinarySearchTree() {
23     this->NIL = nullptr;
24     postorder(this->_root, [](Node *tmp){delete tmp;});
25 }
26
27 // inserting a node in Binary Search Tree
28 template<typename D, class Node>
29 Node* BinarySearchTree<D, Node>::insertNode(double &k, D &d) {
30     Node *z = new Node(k, d);
31
32     auto curr = this->_root;
33     auto prev = this->NIL;
34
35     // going doing checking if must be left or right
36     while(curr != this->NIL){
37         prev = curr;
38         curr = z->getKey() < curr->getKey() ? curr->getLeft() :
39             ↪ curr->getRight();
40     }
41
42     // set parent to previously saved node
43     z->setParent(prev);
44
45     // checking if is left son or right son
46     if(prev == this->NIL) // or root
47         this->_root = z;
48     else if(z->getKey() < prev->getKey())
49         prev->setLeft(z);
50     else
51         prev->setRight(z);
52
53     // new allocated node got NIL as children
54     z->setLeft(this->NIL);
55     z->setRight(this->NIL);
56
57     return z;
58 }
59
60 // delete a node
61 template<typename D, class N>
62 bool BinarySearchTree<D, N>::deleteNode(double &k) {
63     // must search
64     auto z = this->search(k);

```



```

64     if(z == nullptr)
65         return false; // if cannot find return false
66
67     auto y = z;
68
69     if( z->getLeft() == this->NIL )           // if no left son
70         this->transplant(z, z->getRight()); // right goes in Z
71     else if( z->getRight() == this->NIL ) // if no right son
72         this->transplant(z, z->getLeft()); // left goes in Z
73     else {
74         y = this->minimum(z->getRight()); // search for successor
75         if( y->getParent() != z ) { // if Y is not Z father
76             this->transplant(y, y->getRight()); // swap Y with Y->R
77             y->setRight(z->getRight());
78             y->getRight()->setParent(y);
79         }
80         this->transplant(z, y); // swap Z with Y
81         y->setLeft(z->getLeft()); // attach Z-> L to Y
82         y->getLeft()->setParent(y);
83     }
84
85     delete z;
86     return true;
87 }
88
89
90 // transplant a node attaching father
91 template <typename D, class Node>
92 void BinarySearchTree<D, Node>::transplant(Node *u, Node *v) {
93     if( u->getParent() == this->NIL)
94         this->_root = v;
95     else if(u->getParent()->getLeft() == u)
96         u->getParent()->setLeft(v);
97     else
98         u->getParent()->setRight(v);
99
100     // unconditioned assignement since using NIL node
101     v->setParent(u->getParent());
102 }
103
104
105 template <typename D, class Node>
106 Node* BinarySearchTree<D, Node>::search(double &key) {
107     auto tmp = this->_root;
108
109     // while down to NIL or key found
110     while(tmp != this->NIL && tmp->getKey() != key)
111         tmp = key < tmp->getKey() ? tmp->getLeft() : tmp->getRight();
112

```

```

113     return tmp;
114 }
115
116
117 // return minimum node
118 template <typename D, class Node>
119 Node* BinarySearchTree<D, Node>::minimum(Node *a) {
120     auto tmp = a;
121
122     // going down to left
123     while(tmp->getLeft() != this->NIL)
124         tmp = tmp->getLeft();
125     return tmp;
126 }
127
128
129 // return maximum
130 template <typename D, class Node>
131 Node* BinarySearchTree<D, Node>::maximum(Node *a) {
132     auto tmp = a;
133
134     //going down to right
135     while(tmp->getRight() != this->NIL)
136         tmp = tmp->getRight();
137     return tmp;
138 }
139
140
141 template <typename D, class Node>
142 Node* BinarySearchTree<D, Node>::successor(Node *x) {
143     auto tmp = x;
144     auto y = x->getRight();
145
146     // if there is a right tree, return minimum
147     if(y != this->NIL)
148         return minimum(y);
149
150     y = tmp->getParent();
151     // goes up until tmp is left son or root found (=NIL)
152     while(y != this->NIL && y->getRight() == tmp){
153         tmp = y;
154         y = y->getParent();
155     }
156     return y;
157 }
158
159
160 template <typename D, class Node>
161 Node* BinarySearchTree<D, Node>::predecessor(Node *x) {

```

```

162     auto tmp = x;
163     auto y = x->getLeft();
164
165     // if there are left branch, return max
166     if(y != this->NIL)
167         return maximum(y);
168
169     y = tmp->getParent();
170     // goes up until tmp is right son or root found (=NIL)
171     while(y != this->NIL && y->getLeft() == tmp){
172         tmp = y;
173         y = y->getParent();
174     }
175     return y;
176 }
177
178
179 template <typename D, class Node>
180 void BinarySearchTree<D, Node>::inorder(Node *tmp, void(*lambda)(Node*)) {
181     if(tmp != this->NIL) {
182         inorder(tmp->getLeft(), lambda);
183         lambda(tmp); // [Catcher](Tmp){/*implementation*/}
184         inorder(tmp->getRight(), lambda);
185     }
186 }
187
188
189 template <typename D, class Node>
190 void BinarySearchTree<D, Node>::postorder(Node *tmp, void(*lambda)(Node*))
191 ↪ {
192     if(tmp != this->NIL) {
193         lambda(tmp); // [Catcher](Tmp){/*implementation*/}
194         postorder(tmp->getLeft(), lambda);
195         postorder(tmp->getRight(), lambda);
196     }
197 }

```

Codice 1.11: redblacktree.hpp

```

1  #ifndef _REDBLACKTREE_HPP_
2  #define _REDBLACKTREE_HPP_
3
4  #include <trees/binarysearchtree.hpp>
5  #include <nodes/redblacknode.hpp>
6
7
8  /**

```

```

9  * @brief Red Black tree
10 * @details Templated Red Black tree that
11 * inherits from Binary Search tree <Data, RBnode>
12 * @tparam D data
13 */
14 template <typename D, class Node = RedBlackNode<D>>
15 class RedBlackTree : public BinarySearchTree<D, Node>{
16     private:
17         // insert and delete Fix for restoring RB property
18         void insertFix(Node *x);
19         void deleteFix(Node *x);
20
21         // left Rotate and right Rotate for balancing method
22         void leftRotate(Node *x);
23         void rightRotate(Node *x);
24
25     public:
26         RedBlackTree(){};
27         ~RedBlackTree();
28
29         //create and insert RBNode, with rb fixes
30         Node* insertNode(double &k, D &d) override;
31
32         //delete RBNode if exist
33         bool deleteNode(double &k) override;
34
35         // inorder print
36         void print();
37
38         //override for correct print
39         friend std::ostream& operator<<(std::ostream &Str, RedBlackTree<D>
40             ↪ *rb) {
41             rb->print();
42             return Str;
43         };
44
45     };
46 #endif //_REDBLACKTREE_HPP_

```

Codice 1.12: redblacktree.cpp

```

1  #include <trees/redblacktree.hpp>
2  #include <other/debug.hpp>
3
4  #include "binarysearchtree.cpp" // include for templated implementation
5  ↪ workaround

```

```

6
7 // left rotate a node for balancing
8 template <typename D, class Node> // aka move down
9 void RedBlackTree<D, Node>::leftRotate(Node *x) { // O(1) op keeping BST
    ↪ property
10     Node *y = x->getRight(); // set y
11     x->setRight(y->getLeft()); // move l-subtree of y to r-subtree of x
12
13     if ( y->getLeft() != this->NIL )
14         y->getLeft()->setParent(x);
15
16     y->setParent(x->getParent()); // link fathers
17
18     if ( x->getParent() == this->NIL )
19         this->_root = y; // if is X was root
20     else if ( x == x->getParent()->getLeft() )
21         x->getParent()->setLeft(y);
22     else
23         x->getParent()->setRight(y);
24
25     y->setLeft(x);
26     x->setParent(y);
27 }
28
29
30 //rotate a node to right (aka move down) keeping BST property
31 template <typename D, class Node>
32 void RedBlackTree<D, Node>::rightRotate(Node *x) {
33     Node *y = x->getLeft();
34     x->setLeft(y->getRight());
35
36     if ( y->getRight() != this->NIL )
37         y->getRight()->setParent(x);
38
39     y->setParent(x->getParent());
40
41     if ( x->getParent() == this->NIL )
42         this->_root = y;
43     else if ( x == x->getParent()->getLeft() )
44         x->getParent()->setLeft(y);
45     else
46         x->getParent()->setRight(y);
47
48     y->setRight(x);
49     x->setParent(y);
50 }
51
52
53 template <typename D, class Node>

```

```

54 void RedBlackTree<D, Node>::insertFix(Node *x) { //  $O(\log n)$ 
55     Node *y;
56     while(x->getParent()->isRed()) {
57         if(x->getParent() == x->getParent()->getParent()->getLeft()) {
58             y = x->getParent()->getParent()->getRight();
59             if(y->isRed()) { // CASE 1: parent of x and uncle are RED
60                 x->getParent()->setBlack();
61                 y->setBlack(); // change their color and
62                 x->getParent()->getParent()->setRed();
63                 x = x->getParent()->getParent(); // move point to grand father
64             } else { // CASE 2: uncle is BLACK
65                 if(x == x->getParent()->getRight()) { // x is right son
66                     x = x->getParent(); // move upper
67                     leftRotate(x); // then rotate so x became left son
68                 } // CASE 3: // uncle is BLACK but x is left son
69                 x->getParent()->setBlack();
70                 x->getParent()->getParent()->setRed();
71                 rightRotate(x->getParent()->getParent()); // compression aka
72                     ↪ same BH
73             }
74         } else {
75             y = x->getParent()->getParent()->getLeft();
76             if(y->isRed()) { // CASE 1: parent of x and uncle are RED
77                 x->getParent()->setBlack();
78                 y->setBlack(); // change their color and
79                 x->getParent()->getParent()->setRed();
80                 x = x->getParent()->getParent(); // move point to grand father
81             } else { // CASE 2: uncle is BLACK
82                 if(x == x->getParent()->getLeft()) { // x is left son
83                     x = x->getParent(); // move upper
84                     rightRotate(x); // then rotate so x became right son
85                 } // CASE 3: // uncle is BLACK but x is left son
86                 x->getParent()->setBlack();
87                 x->getParent()->getParent()->setRed();
88                 leftRotate(x->getParent()->getParent()); // compression aka same
89                     ↪ BH
90             }
91         }
92     }
93     this->_root->setBlack();
94 }
95
96 template <typename D, class Node>
97 Node* RedBlackTree<D, Node>::insertNode(double &k, D &d) {
98     auto z = BinarySearchTree<D, Node>::insertNode(k, d); // BST insert
99     z->setRed(); // set flag as red
100    this->insertFix(z); // fix property
101    return z;

```

```

101 }
102
103
104 template <typename D, class Node>
105 void RedBlackTree<D, Node>::deleteFix(Node *x) {
106     Node *w;
107
108     while( x != this->_root && x->isBlack() ) {
109         if( x == x->getParent()->getLeft() ) {
110             w = x->getParent()->getRight();
111
112             if ( w->isRed() ) { //////////
113                 w->setBlack(); ////////// CASE
114                 x->getParent()->setRed(); //// 1
115                 leftRotate(x->getParent());
116                 w = x->getParent()->getRight();
117             }
118
119             if ( w->getLeft()->isBlack() && w->getRight()->isBlack() ) {
120                 w->setRed(); ////////// CASE
121                 x = x->getParent(); ////////// 2
122             }
123             else { //////////
124                 if ( w->getRight()->isBlack() ) {
125                     w->getLeft()->setBlack(); //// CASE
126                     w->setRed(); ////////// 3
127                     rightRotate(w); //////////
128                     w = x->getParent()->getRight();
129                 }
130                 ////////// CASE
131                 w->setColor(x->getParent()->getColor());
132                 x->getParent()->setBlack(); ////
133                 w->getRight()->setBlack(); //// 4
134                 leftRotate(x->getParent()); ////
135                 x = this->_root; //////////////////
136             }
137         }
138         else { // same as upper code but swapped left and right
139             w = x->getParent()->getLeft();
140
141             if ( w->isRed() ) { //////////
142                 w->setBlack(); ////////// CASE
143                 x->getParent()->setRed();//////
144                 rightRotate(x->getParent());/// 1
145                 w = x->getParent()->getLeft();//
146             }
147
148             if ( w->getRight()->isBlack() && w->getLeft()->isBlack() ) {
149                 w->setRed(); ////////// CASE

```

```

150         x = x->getParent();///// 2
151     }
152     else {
153         if ( w->getLeft()->isBlack() ) { //
154             w->getRight()->setBlack(); /////
155             w->setRed(); ////////////////// CASE
156             leftRotate(w); /////////// 3
157             w = x->getParent()->getLeft();//
158         }
159         ////////////////// CASE 4
160         w->setColor(x->getParent()->getColor());
161         x->getParent()->setBlack();
162         w->getLeft()->setBlack();
163         rightRotate(x->getParent());
164         x = this->_root;
165     }
166 }
167 }
168 x->setBlack(); // (Cormen pag. 270)
169 }
170
171
172 // delete a node if exist and fixup
173 template <typename D, class Node>
174 bool RedBlackTree<D, Node>::deleteNode(double &k) {
175     auto z = this->search(k);
176
177     auto y = z;
178     auto y_original_color = y->getColor();
179
180     if(z == this->NIL)
181         return false;
182
183     Node *x;
184
185     // when deleting z, y will be the "successor"
186     // so must check if is respecting RB property
187     // saving his color
188     if( y->getLeft() == this->NIL ) {
189         x = z->getRight();
190         this->transplant(z, z->getRight());
191     }
192     else if( z->getRight() == this->NIL ) {
193         x = z->getLeft();
194         this->transplant(z, z->getLeft());
195     }
196     else {
197         // y has no left child
198         y = this->minimum(z->getRight());

```



```

199     y_original_color = y->getColor();
200
201     // we track the color of new positional node
202     x = y->getRight();
203
204     if( y->getParent() == z ) //y righson of z
205         x->setParent(y);
206     else { // must inherit right son of z
207         this->transplant(y, y->getRight());
208         y->setRight(z->getRight());
209         y->getRight()->setParent(y);
210     }
211
212     // inherit left son of z
213     this->transplant(z, y);
214     y->setLeft(z->getLeft());
215     y->getLeft()->setParent(y);
216
217     // y is the old z and so got same color
218     y->setColor(z->getColor());
219 }
220
221 // fix black root
222 // fix two red nodes
223 // fix Black High lose
224 if(y_original_color == BLACK)
225     deleteFix(x);
226
227
228 delete z;
229 return true;
230 }
231
232
233 template <typename D, class Node>
234 void RedBlackTree<D, Node>::print() {
235     this->inorder(this->_root, [](Node *tmp) {
236         std::cout<<tmp; });
237 }

```

Codice 1.13: redblackhash.hpp

```

1  #ifndef _REDBLACKHASH_HPP_
2  #define _REDBLACKHASH_HPP_
3
4  #include <trees/redblacktree.hpp>
5  #include <hashtables/hashtable.hpp>

```

```

6
7  /**
8   * @brief Red Black Hash
9   * @details data structure that use a Red Black to stores in every node
10  * an Hashtables.
11  * Key 1 is the int used to indexes a red black, Key 2 is the int to
12  ↪ indexes
13  * a single cell of an hashtable. D d is the data to store in hashtables
14  * @tparam D data to store in hashtables
15  */
16 template <typename D>
17 class RedBlackHash {
18     private:
19         RedBlackTree<HashTable<D> *> *_rb;
20         int _size;
21
22     public:
23         //fixed size of 285700
24         RedBlackHash(int s = 285700);
25         ~RedBlackHash(){};
26
27         //redblack getter
28         RedBlackTree<HashTable<D> *> *getRBHashTree(){return this->_rb;};
29
30         bool insert(int k1, int k2, D d);
31         RedBlackNode<HashTable<D> *>* search(int k1, int k2, D d);
32         bool remove(int k1, int k2, D d);
33
34         // print chaining
35         friend std::ostream& operator<<(std::ostream &Str, RedBlackHash<D>
36         ↪ *rh) {
37             Str<<rh->getRBHashTree()<<"\n";
38             return Str;
39         };
40
41 #endif // _REDBLACKHASH_HPP_

```

Codice 1.14: redblackhash.cpp

```

1 #include <iostream>
2
3 #include <redblackhash.hpp>
4
5 #include "hashtables/hashtable.cpp"
6 #include "trees/redblacktree.cpp"

```

```

7
8
9  template <typename D>
10 RedBlackHash<D>::RedBlackHash(int size) : _size(size) {
11     this->_rb = new RedBlackTree<HashTable<D> *>;
12 }
13
14
15 template <typename D>
16 bool RedBlackHash<D>::insert(int k1, int k2, D d) {
17     auto key1 = static_cast<double>(k1);
18     auto key2 = static_cast<double>(k2);
19
20     auto node = this->_rb->search(key1);
21
22     if(node == this->_rb->getNil()) {           // if it's a new node
23         auto hashtable = new HashTable<D>(this->_size); // allocate hashtable
24         node = this->_rb->insertNode(key1, hashtable); // insert this new
25         ↪ hashtable
26         node->getData()->insert(key2, d);       // insert string
27         return true;
28     }
29     else
30         if(node->getData()->search(key2) < 0)    // if there isnt
31         ↪ already a key2
32         if(node->getData()->insert(key2, d) >= 0) // and also there is
33         ↪ space
34         return true;
35     else
36         std::cerr<<"\nerror: there is no space left in hash table in node
37         ↪ "<<key1<<"\n";
38     else
39         std::cerr<<"\nerror: there is already an hashnode with key =
40         ↪ "<<key2<<"\n";
41
42     // if here means that there is already a key or there is no space
43     return false;
44 }
45
46
47 template <typename D>
48 RedBlackNode<HashTable<D> *> RedBlackHash<D>::search(int k1, int k2, D d)
49 ↪ {
50     auto key1 = static_cast<double>(k1);
51     auto key2 = static_cast<double>(k2);
52
53     auto node = this->_rb->search(key1);
54
55     if(node != this->_rb->getNil()) {

```

```

50     auto j = node->getData()->search(key2);
51     if( j >= 0 ) {
52         if(node->getData()->search(j) == d )
53             return node;
54         else
55             std::cerr<<"\nerror: no value in node["<<key1<<"]
56             ↪ hash["<<key2<<"]-> "<<d<<"\n"<<
57             ↪ "maybe you were looking for:
58             ↪ "<<node->getData()->search(j)<<"\n";
59     }
60     else{
61         std::cerr<<"\nerror: no hash node with key2 = ["<<k2<<"]\n";
62         std::cerr<<"\nhash in node["<<key1<<"] "<<"only got keys: ";
63         node->getData()->printKeys();
64     }
65     }
66     else
67         std::cerr<<"\nerror: no RB node with key1 = ["<<k1<<"]\n";
68     }
69     return nullptr;
70 }
71
72 template <typename D>
73 bool RedBlackHash<D>::remove(int key1, int key2, D data) {
74     // if tuple is correct
75     auto node = this->search(key1, key2, data);
76     if(node != nullptr) {
77         auto k1 = static_cast<double>(key1);
78         auto k2 = static_cast<double>(key2);
79
80         // if hashtable got only one value (key1)
81         if(node->getData()->getQuantity() == 1)
82             this->_rb->deleteNode(k1); // delete RB node
83         else
84             node->getData()->remove(k2); // just remove value
85
86         return true;
87     }
88     return false;
89 }

```

Codice 1.15: parser.hpp

```

1  #ifndef _PARSER_HPP_
2  #define _PARSER_HPP_
3

```

```

4  #include <string>
5  #include <redblackhash.hpp>
6
7  /**
8   * @brief Parser for Red Black Hashtable
9   * @details Used to fill from a file an hashtable.
10  * Also providing method for C.R.D. (create, retrieve, delete) tuples
11  * tuples must be in format <int> keys and <string> data ->
12  ↪ key1:key2:data
13  */
14  class Parser {
15  private:
16      int _hashsize;
17      std::string _filename;
18      RedBlackHash<std::string> *_rh;
19
20  public:
21      Parser(const std::string fn, int hashsize = 9973);
22      ~Parser(){};
23
24      // tuples must be in format <int> keys and <string> data ->
25      ↪ key1:key2:data
26      bool insert(std::string str);
27
28      // tuples must be in format <int> keys and <string> data ->
29      ↪ key1:key2:data
30      bool search(std::string str);
31
32      // tuples must be in format <int> keys and <string> data ->
33      ↪ key1:key2:data
34      bool remove(std::string str);
35
36      //printing data structure
37      void print();
38  };
39
40  #endif // _PARSER_HPP_

```

Codice 1.16: parser.cpp

```

1  #include <parser.hpp>
2  #include <redblackhash.hpp>
3  #include "redblackhash.cpp"
4
5  #include <sstream>
6  #include <fstream>
7

```

```

8  // Parser
9  Parser::Parser(const std::string fn, int hashsize) :
10     // set hashsize
11     _hashsize(hashsize),
12     // set filename
13     _filename(fn),
14     // allocate new RBHash
15     _rh(new RedBlackHash<std::string>(this->_hashsize)) {
16
17     std::ifstream filestream(this->_filename);
18     std::string line;
19
20     if(filestream.fail()){
21         std::cerr << "Error opening file" << std::endl;
22         exit(EXIT_FAILURE);
23     }
24
25     while(std::getline(filestream, line))
26         this->insert(line);
27
28     filestream.close();
29 }
30
31
32 // insert a tuple
33 bool Parser::insert(std::string str) {
34     std::string key, data;
35     std::stringstream tmpstream(str);
36
37     int key1 = 0, key2 = 0;
38
39     if(!std::getline(tmpstream, key, ':')) {
40         std::cerr << "no delimiter key 1" << std::endl;
41         return false;
42     }
43     key1 = atoi(key.c_str());
44
45     if(!std::getline(tmpstream, key, ':')) {
46         std::cerr << "no delimiter key 2" << std::endl;
47         return false;
48     }
49     key2 = atoi(key.c_str());
50
51     if(!std::getline(tmpstream, data)) {
52         std::cerr << "no data" << std::endl;
53         return false;
54     }
55
56     return this->_rh->insert(key1, key2, data);

```

```

57 }
58
59
60 // search a tuple
61 bool Parser::search(std::string str) {
62     std::string key, data;
63     std::stringstream tmpstream(str);
64
65     int key1 = 0, key2 = 0;
66
67     if(!std::getline(tmpstream, key, ':')) {
68         IFDEBUG debug_print << "no delimiter key 1" << std::endl;
69         return false;
70     } key1 = atoi(key.c_str());
71
72     if(!std::getline(tmpstream, key, ':')) {
73         IFDEBUG debug_print << "no delimiter key 2" << std::endl;
74         return false;
75     } key2 = atoi(key.c_str());
76
77
78     if(!std::getline(tmpstream, data)) {
79         IFDEBUG debug_print << "no data" << std::endl;
80         return false;
81     }
82
83
84     return this->_rh->search(key1, key2, data);
85 }
86
87
88 bool Parser::remove(std::string str) {
89     std::string key, data;
90     std::stringstream tmpstream(str);
91
92     int key1 = 0, key2 = 0;
93
94     if(!std::getline(tmpstream, key, ':')) {
95         std::cerr << "no delimiter key 1" << std::endl;
96         return false;
97     } key1 = atoi(key.c_str());
98
99     if(!std::getline(tmpstream, key, ':')) {
100         std::cerr << "no delimiter key 2" << std::endl;
101         return false;
102     } key2 = atoi(key.c_str());
103
104
105     if(!std::getline(tmpstream, data)) {

```

```
106     std::cerr << "no data" << std::endl;
107     return false;
108 }
109
110     return this->_rh->remove(key1, key2, data);
111 }
112
113 void Parser::print(){
114     auto r = this->_rh->getRBHashTree()->getRoot();
115     IFDEBUG {
116         if(r != this->_rh->getRBHashTree()->getNil())
117             std::cout<<"Root:\n"<<this->_rh->getRBHashTree()->getRoot()<<"\n";
118     }
119     std::cout<<"\n"<<this->_rh<<"\n";
120 }
```

Parte II

Capitolo 2

Viaggi Galattici

2.1 Descrizione problema

Il quesito prevede di sviluppare un programma in C++, in grado di trovare in un **grafo non orientato** il **percorso più breve** tra due nodi specifici, ovvero quel percorso tale per cui la somma dei costi associati all'attraversamento degli archi che collegano un punto A ad un punto B è minima ¹. Il problema prevede la possibilità di poter usare alcuni nodi speciali, detti **wormholes**: ogni wormhole nel grafo è collegato ad ogni altro wormhole, inoltre il costo di percorrenza wormhole - wormhole ha peso 1.

2.2 Descrizione strutture dati

2.2.1 Grafi

Le informazioni circa il nome, ovvero la chiave numerica usata come identificativo univoco, ed eventuali dati satelliti, sono salvate in una struttura dati **nodo**, o **vertice**. Per salvare i percorsi dei cammini minimi, ogni vertice mantiene un riferimento al nodo precedente nel cammino. La struttura dati in cui vengono salvati i nodi facenti parte del problema, è un **grafo non orientato**. Per implementare tale struttura dati si è scelto di mantenere per ogni vertice un listato contenente un riferimento agli altri nodi adiacenti ed il relativo peso numerico atto a riportare il costo effettivo dell'attraversamento. Nel caso del grafo non orientato per ogni arco inserito non vi è distinzione tra **arco uscente** o **arco entrante**, per cui si andrà ad inserire due volte volte il sopracitato arco: un arco di peso w tra A e B equivale ad un arco da A a B e un altro da B ad A.

¹ Definizione formale del **cammino minimo** nella teoria dei Grafi

2.2.2 Coda di priorità

La coda di priorità è una particolare coda il cui criterio di inserimento dei vari elementi che compongono la coda è dato non più dall'ordine FIFO ², ma dalla priorità associata ad ogni elemento. Nel caso della **coda a priorità minima**, gli elementi con priorità minima saranno inseriti all'inizio. L'operazione di ricerca del minimo, viene eseguita in $O(1)$, il che rende particolarmente utile la coda di priorità nelle applicazioni che fanno un grande uso della suddetta operazione. Tale ADT³, è usata nell'algoritmo di **Dijkstra**, per la rapida estrazione degli elementi con minor distanza dalla sorgente. ⁴

2.2.3 Min Heap Binario

La struttura dati su cui si basa la coda a priorità minima, sebbene non sia la più efficiente, è l'**heap binario**, una struttura dati basata su albero binario completo sviluppato come vettore che gode della **proprietà heap**: nel caso del **min heap** *il padre di un nodo ha come chiave un valore minore di quella del figlio sinistro e destro*. È stata scelta questa struttura dati rispetto ad un **Heap di Fibonacci**, nonostante abbia complessità $O(\log n)$ nelle operazioni di estrazione del minimo, inserimento e abbassamento priorità, poiché di facile implementazione.

²FIFO: first in, first out.

³ADT: abstract data type.

⁴Algoritmo per il calcolo dei cammini minimi da sorgente unica.

2.3 Formato di input e di output

2.3.1 Input

I dati in input del problema sono:

- V: numero intero di Vertici del grafo
- E: numero intero di Archi del grafo
- W: numero intero di Wormholes presenti nel grafo
- Tuple rappresentante archi: NodoA, NodoB, Peso ⁵

Tali dati sono immagazzinati in un file di testo non binario contenente nel primo rigo i primi tre dati elencati, mentre nei successivi sono presenti le **tuple**. Per rappresentare i wormhole il programma prende gli **ultimi W NodiB** contenuti nel file e li va a salvare in un vettore di vertici.

2.3.2 Output

Il programma sviluppato restituisce in output i nodi che collegano la coppia **sorgente - destinazione** nel minor "tempo" possibile e il relativo costo di tale cammino minimo, **se esiste**: può capitare, come vedremo nel paragrafo "*Test effettuati*" par. 2.6.1, che il grafo non sia connesso e che il nodo destinazione sia raggiungibile solo attraverso i vertici di tipo wormhole. Inoltre il programma restituisce, il cammino minimo (vertici da attraversare e costo totale) facendo uso dei nodi speciali wormhole. L'output secondario può mancare nel caso in cui non si incontrino wormhole, oppure il wormhole di partenza è uguale a quello di destinazione.

⁵ndr: NodoA e NodoB sono le chiavi intere dei vertici e Peso è un intero usato per rappresentare il peso di tale arco.

2.4 Descrizione algoritmo

2.4.1 Pseudo codice

Per calcolare i cammini minimi⁶ da una sorgente si è scelto di utilizzare l'algoritmo greedy proposto dall'olandese **Edsger Dijkstra** che va a scegliere localmente un nodo adiacente più vicino a quello analizzato. Per definizione della sottostruttura ottima di un cammino minimo, il **primo wormhole** aggiunto nell'albero dei cammini minimi, è quello che si può raggiungere più velocemente da una data sorgente. Per tanto è stato modificato Dijkstra per salvare i wormhole che incontra durante la creazione del cammino, e per fermarsi una volta raggiunta la destinazione.

L'algoritmo che verrà mostrato di seguito è il cuore del programma: in input riceve i nodi sorgente e destinazione e in fase di elaborazione restituisce **i cammini minimi**. È stato usato il plurale in quanto, potrebbe esserci un secondo cammino che fa uso dei wormhole, o viceversa se il grafo non è connesso, esserci solamente il cammino con i wormhole.

La procedura **Galactic Dijkstra** applica una prima volta Dijkstra dalla sorgente fino a che non trova la destinazione e nel mentre salva tutti i wormhole che incontra, estraendone solo il primo (riga 1). Se esiste un wormhole nell'albero dei cammini minimi radicato in S, allora si procede ad una seconda applicazione di Dijkstra, usando come sorgente il nodo di destinazione. Se questi due wormhole sono diversi allora si calcola il percorso minimo tra i due e si aggiunge un arco simbolico di peso 1.

⁶Anche definiti come albero dei cammini minimi radicati in una sorgente.

Algorithm: Galactic Dijkstra

```

input : Source node  $\in V$ , Destination node  $\in V$ 
result: print fast path from  $s$  to  $d$ , w/ and w/o
          wormholes if any
1  $w_1 = \text{apply Dijkstra from } S \text{ and save first wormhole}$ 
    $\text{encountered};$ 
2  $\text{distance} = \text{printPath}(s, d);$ 
3 if  $\exists w_1 :$ 
4    $w_2 = \text{apply Dijkstra from } D \text{ and save first wormhole}$ 
     $\text{encountered};$ 
5   if  $\exists w_2$  and  $w_1 \neq w_2 :$ 
6      $dw_1 = \text{printPath}(s, w_1);$ 
7      $dw_2 = \text{printPath}(w_2, d);$ 
8      $d_{dw_1+dw_2} = dw_1 + 1 + dw_2;$ 

```

2.4.2 Diagrammi delle classe e dettagli architetturali

Priority Queue e Heap

La coda di priorità è stata sviluppata come detto in precedenza facendo uso di un **Heap Binario**, per la precisione un Min Heap. Si poteva creare una classe generica priority queue e usare un **pattern comportamentale** (e.g. Strategy) per sfruttare la possibilità di cambiare comportamento (minima priorità o massima) in base ad un flag in fase di creazione della classe. Non è stato usato tale approccio in quanto nell'algoritmo di Dijkstra si fa uso solamente di una coda a minima priorità.

Un'altra nota riguarda l'uso della classe Min Heap all'interno della Priority Queue: sarebbe stato utile usare un' **interfaccia** (i.g. classe astratta) per l'heap dando la possibilità al programmatore di usare un'altra tipologia (e.g.: Fibonacci, Brodal, Binomiale, etc.). Non avendo usato tale approccio la classe Min Priority Queue è dipendente dall' Heap Binario.

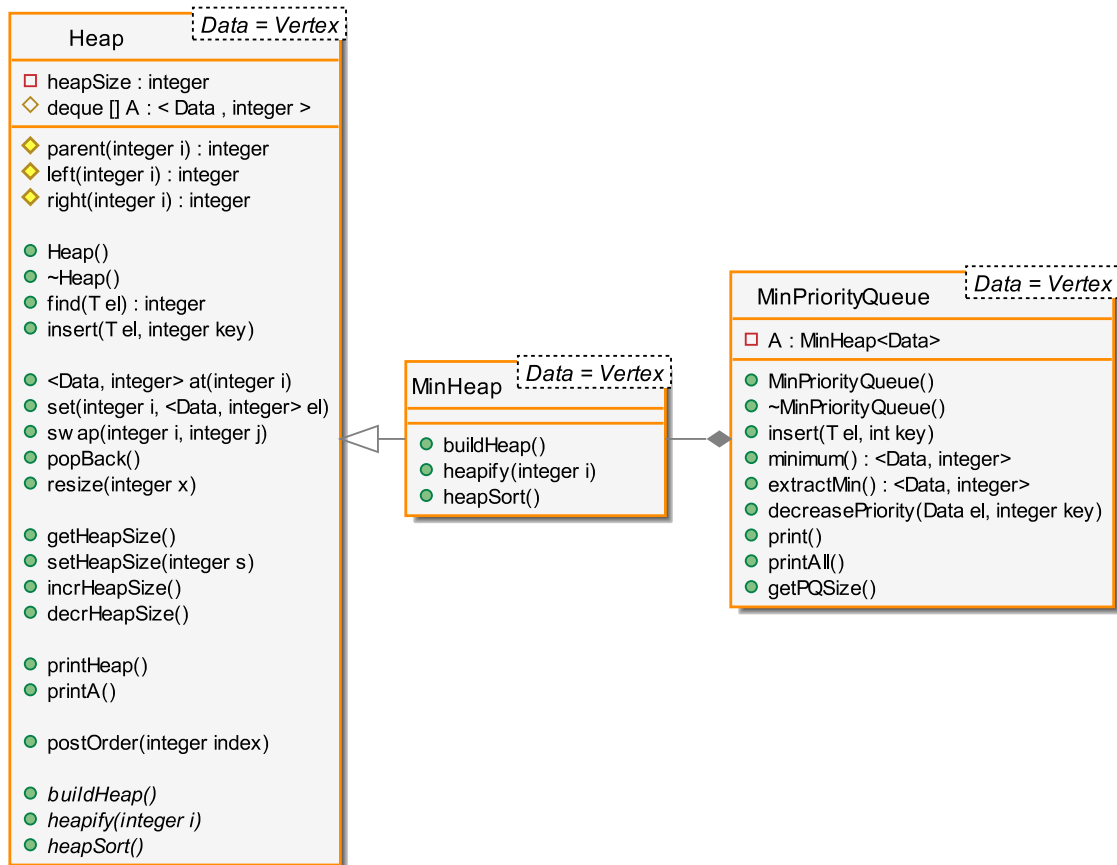


Figura 2.1: Min priority Queue

N.d.r.: ogni classe ha un parametro template Data, in questo caso specifico viene usato un puntatore ad oggetto Vertex, mostrato di seguito.

Grafo e Vertici

Ogni elemento del grafo, i **vertici**, sono dei nodi che ereditano da un generico "oggetto" **Item** la possibilità di inserire Dati indentificati da una chiave. A tal proposito non avendo bisogno di conservare nessun dato si è deciso di usare come parametro del template un **puntatore a void**. Ogni vertice ha una mappatura con i vertici **adiacenti** (realizzata tramite un hashtable di tipo unordered), un riferimento al padre nell' albero dei cammini minimi, e la distanza dal nodo radice.

Il **grafo** possiede un vettore di puntatori a vertici e ha metodi per creare un albero dei cammini minimi (Dijkstra), restituire o stampare il percorso da una sorgente e una destinazione. Il metodo `dijkstra` è stato ridefinito in modo tale da poter eseguire operazioni aggiuntive alla fine del rilassamento di un nodo estratto dalla coda: si può decidere di usare una funzione lambda oppure un puntatore a funzione per inserire un **criterio di stop** nell'algoritmo (e.g. raggiunto un nodo specifico), inoltre verrà restituito l'ultimo elemento estratto.

N.d.r.: il metodo `dijkstra` internamente fa uso delle subroutine *initSingleSource* e *relax* come da manuale⁷ ma poichè tale implementazione può cambiare (non far uso della Min Priority Queue ad esempio) si è deciso di non inserirle come metodi privati durante la definizione della classe. Per cui verranno citate e mostrate solo per far capire la connessione con la min priority queue.

⁷Vedere riferimenti bibliografici.

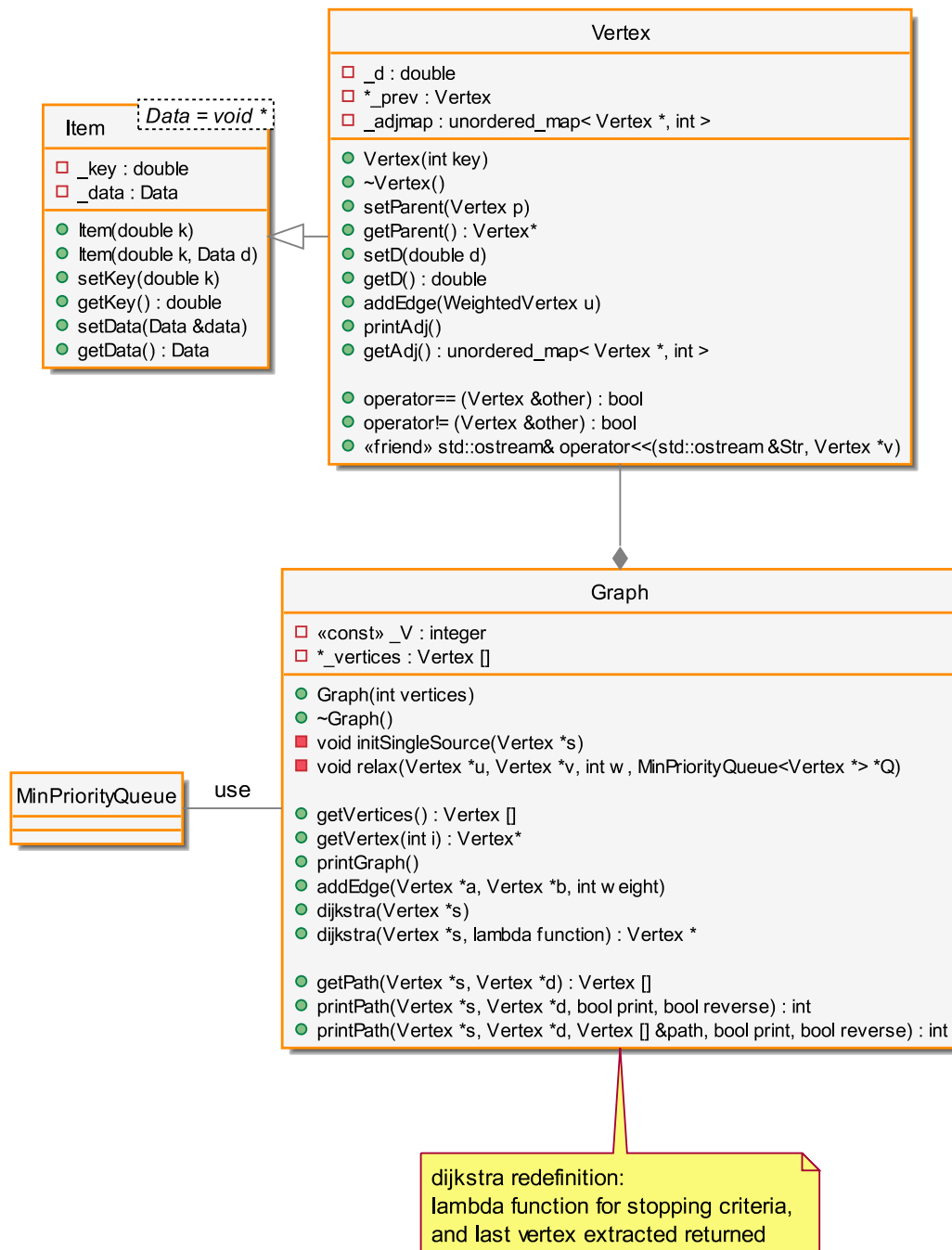


Figura 2.2: Vertici e Grafo

Parser e Grafo galattico

Il main del programma userà la classe **Loader** per creare e istanziare correttamente un oggetto di tipo **GalacticGraph**. Tale classe altro non è che una specializzazione del grafo base, con l'aggiunta di una mappatura dei wormhole del grafo caricato dal Loader e dell'algoritmo Galactic Dijkstra (par. 2.4.1).

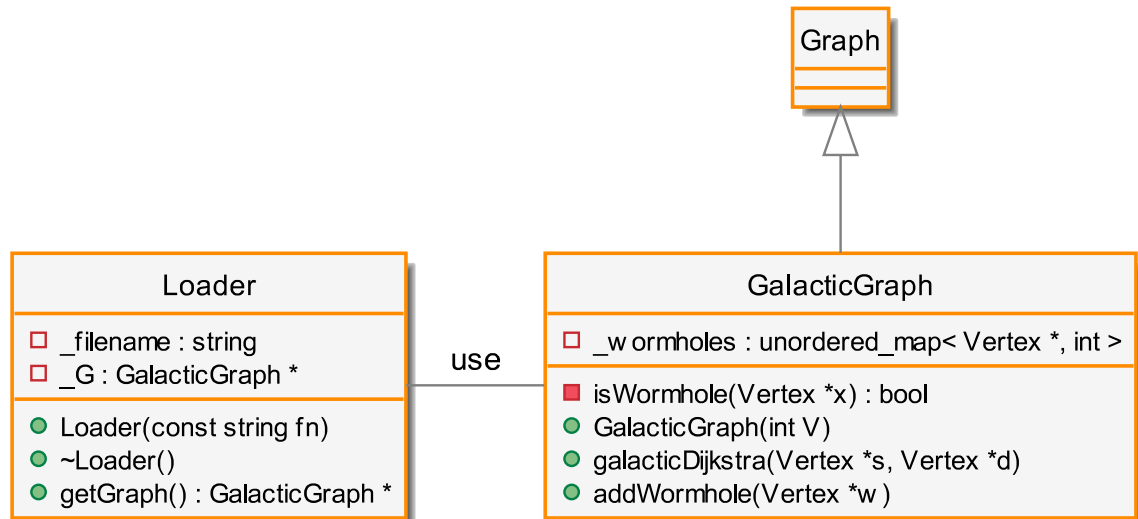


Figura 2.3: Loader e Grafo galattico

2.5 Studio complessità

La complessità di tempo data dall'algoritmo galactic dijkstra è influenzata dalle **due esecuzioni** (al più) del classico algoritmo di Dijkstra. Il calcolo dei percorsi impiega $O(3E)$ poiché per raggiungere un nodo fino alla sorgente vuol dire ripercorrere l'albero attraversando ogni arco nel caso peggiore, quindi mettendoci un tempo lineare. Per cui si considerano le due applicazioni di Dijkstra che impiegano $O(2(V + E)2\log_2 V)$ che diventa $O(2E2\log_2 V)$ se ogni vertice è raggiungibile dalla sorgente, ma poichè è possibile portare le costanti moltiplicative fuori diremo che la complessità finale sarà $\mathbf{O(E\log_2 V)}$. L'algoritmo di Dijkstra impiega tale complessità poiché influenzato dalle operazioni *extractMin()* e *decreaseKey()* impiegate $|E|$ volte dalla coda di min priorità basata su min heap. Si poteva pensare di usare un Heap di Fibonacci poiché si è notato che l'algoritmo effettua più operazioni di *decreaseKey()* che di *extractMin()*, e nella struttura dati citata tale operazioni hanno rispettivamente costo computazionale $O(1)$ e $O(\log_2 V)$. Ciò avrebbe comportato un miglioramento nel caso in cui ci fossero stati molti archi, avendo un costo asintotico pari a $O(V\log_2 V + E)$.

2.6 Test e risultati

2.6.1 Test effettuati

I risultati estratti da una shell dopo aver eseguito il programma prevedono vari grafi. Per ognuno di essi vi sarà presente il grafico relativo.

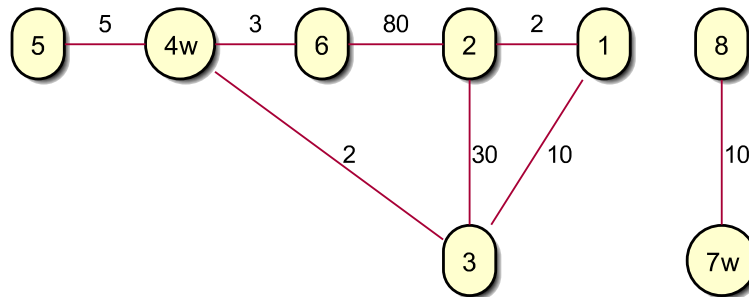


Figura 2.4: Grafo non connesso.

Filling a graph with 8 nodes, 8 edges, 2 wormholes

STARTING READING FILE: 6

STARTING READING 2ndhalf: 2

Looking **for** a path from 1 to 8

Travel without wormholes: there are no paths that connect 1 with 8
(disconnected graph)

Using wormholes -> [4, 7]: 1-3-4^7-8 in 23 unit **time**

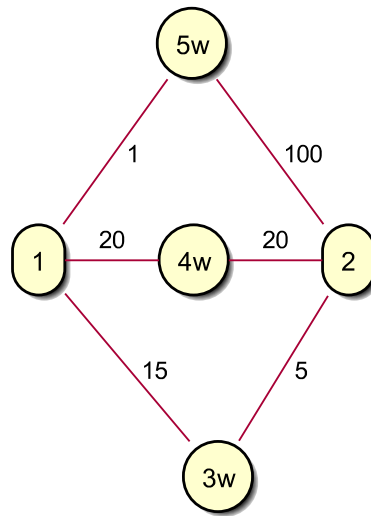


Figura 2.5: Grafo a diamante.

Filling a graph with 5 nodes, 6 edges, 3 wormholes

STARTING READING FILE: 3

STARTING READING 2ndhalf: 3

Looking **for** a path from 1 to 2

Travel without wormholes: 1-3-2 in 20 **time** unit

Using wormholes -> [5, 3]: 1-5^3-2 in 7 unit **time**

Nel grafo seguente il percorso con wormhole non è mostrato in quanto il wormhole usato dalla sorgente e dalla destinazione è lo stesso, infatti il nodo 11 è presente in entrambi i cammini minimi.

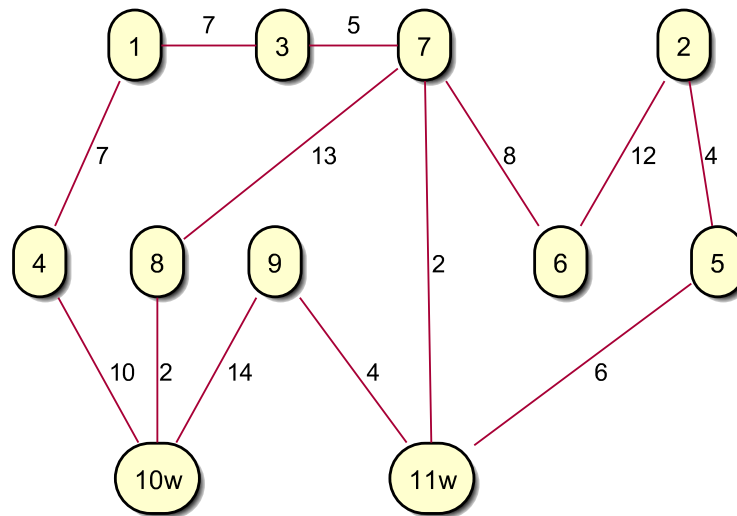


Figura 2.6: Grafo d'esempio.

Filling a graph with 11 nodes, 13 edges, 2 wormholes

STARTING READING FILE: 11

STARTING READING 2ndhalf: 2

Looking **for** a path from 1 to 2

Travel without wormholes: 1-3-7-11-5-2 in 24 **time** unit

No fast travel with wormhole.

Per puro stress testing del programma, si è utilizzato un file contenente 317080 nodi, 1049866 archi e 10 wormholes, ovviamente previo commento nell'implementazione del parser circa i controlli sull'input. Successivamente si usa lo stesso file ma senza wormhole in modo da non far applicare una seconda volta dijkstra. Il risultato mostrato di seguito contiene l'esecuzione calcolandone i tempi tramite l'applicativo *time* presente nei sistemi operativi **NIX*

```
Filling a graph with 317080 nodes, 1049866 edges, 10 wormholes
```

```
STARTING READING FILE: 1049856
```

```
STARTING READING 2ndhalf: 10
```

```
Looking for a path from 316972 to 317029
```

```
Travel without wormholes:
```

```
316972-264079-298841-63506-59115-5530-26742-65338
```

```
-191800-287822-125483-121155-107180-308349-317029
```

```
in 104 time unit
```

```
Using wormholes -> [316972, 317029]: 316972^317029 in 1 unit time
```

```
real    1m42.700s
```

```
user    1m42.313s
```

```
sys     0m0.188s
```

Da notare come il file senza wormhole ci impiega circa la **metà**.

```
Filling a graph with 317080 nodes, 1049866 edges, 0 wormholes
```

```
STARTING READING FILE: 1049866
```

```
STARTING READING 2ndhalf: 0
```

```
Looking for a path from 316972 to 317029
```

```
Travel without wormholes:
```

```
316972-264079-298841-63506-59115-5530-26742-65338
```

```
-191800-287822-125483-121155-107180-308349-317029
```

```
in 104 time unit
```

```
No wormhole from Source to Destination.
```

```
real    0m56.303s
```

```
user    0m56.047s
```

```
sys     0m0.094s
```


2.7 Codice sorgente

Codice 2.17: heap.hpp

```

1  #ifndef _HEAP_HPP_
2  #define _HEAP_HPP_
3
4  #include <iostream>
5  #include <algorithm>
6  #include <utility>
7  #include <deque>
8
9  /** Heap Data Structure */
10 template <class T> // object inside this DS are templates
11 class Heap {
12     private:
13         int heapSize;
14
15     protected: // made attributes protected 'cause
16                 // we need in Priority Queues
17         std::deque< std::pair<T, int> > A;
18
19     public:
20         Heap() : heapSize(0){};
21         ~Heap(){};
22
23         int find(T el);
24
25         // inserting elements using std::pair
26         void insert(T el, int key); // must use Int for key and T for obj to
27                                     ↪ store.
28
29         int parent(int i){return (i-1)/2;};
30         int left(int i){return (2*i + 1);};
31         int right(int i){return (2*i + 2);};
32
33     /**
34      * override of deque positional operators
35      * must use this for access at deque (e.g. Priority Queue)
36      * return std::pair from position at i
37      */
38     std::pair<T, int> at(int i);
39
40     /**
41      * override of deque positional operators
42      * must use this for access at deque (e.g. Priority Queue)
43      * set pair<key, T element> at position i

```

```

44     */
45     void set(int i, std::pair<T, int> el);
46
47     /**
48      * override of deque positional operators
49      * must use this for access at deque (e.g. Priority Queue)
50      * swap elements using index (wrapper to std::iter_swap)
51      */
52     void swap(int i, int j);
53
54     /**
55      * override of deque positional operators
56      * must use this for access at deque (e.g. Priority Queue)
57      * wrapper to pop_back()
58      */
59     void popBack();
60
61     /**
62      * override of deque positional operators
63      * must use this for access at deque (e.g. Priority Queue)
64      * wrapper to resize()
65      */
66     void resize(int x);
67
68     int getHeapSize(){return heapSize;}
69     void setHeapSize(int s){heapSize = s;}
70
71     void incrHeapSize(){heapSize++;}
72     void decrHeapSize(){heapSize--;}
73
74     /** Print the heap using heapSize as upperbound */
75     void printHeap();
76
77     /** Print all the A deque */
78     void printA();
79
80     /** Virtual method to implement in max Heap and min heap */
81     virtual void buildHeap() = 0; // pure specifier = 0, same as {}
82
83     /** Virtual method to implement in max Heap and min heap */
84     virtual void heapify(int i) = 0; // pure specifier = 0, same as {}
85
86     /** Virtual method to implement in max Heap and min heap */
87     virtual void heapSort() = 0; // pure specifier = 0, same as {}
88
89     /** Print in postOrder starting from index */
90     void postOrder(int index);
91 };
92

```

```

93
94 #endif //_HEAP_HPP_

```

Codice 2.18: heap.cpp

```

1  #include <priorityqueue/heap/heap.hpp>
2
3  template <class T>
4  void Heap<T>::insert(T el, int key) {
5      this->A.push_back( std::make_pair(el, key) );
6  }
7
8  template <class T>
9  std::pair<T, int> Heap<T>::at(int i) {
10     return this->A.at(i);
11 }
12
13 template <class T>
14 void Heap<T>::set(int i, std::pair<T, int> el) {
15     this->A.at(i) = el;
16 }
17
18 template <class T>
19 void Heap<T>::swap(int i, int j){
20     std::iter_swap(this->A.begin()+i, this->A.begin()+j);
21 }
22
23 template <class T>
24 void Heap<T>::popBack(){
25     this->A.pop_back();
26 }
27
28 template <class T>
29 void Heap<T>::resize(int x){
30     this->A.resize(x);
31 }
32
33
34 template <class T>
35 void Heap<T>::postOrder(int index){
36     if(index < Heap<T>::getHeapSize()){
37         postOrder(Heap<T>::left(index));
38         postOrder(Heap<T>::right(index));
39         std::cout << this->A.at(index).first << " ";
40     }
41 }
42

```

```

43
44 template <class T>
45 void Heap<T>::printHeap(){
46     for(auto i=0; i < heapSize; i++) {
47         std::cout << "[" << this->A.at(i).second << "]";
48         std::cout << this->A.at(i).first << " ";
49     }
50     std::cout << std::endl;
51 }
52
53
54 template <class T>
55 void Heap<T>::printA(){
56     for(auto i=0; i < this->A.size(); i++){
57         std::cout << "[" << this->A.at(i).second << "]";
58         std::cout << this->A.at(i).first << " ";
59     }
60     std::cout << std::endl;
61 }
62
63
64 // return index
65 template <class T>
66 int Heap<T>::find(T el) {
67     auto it = std::find_if( this->A.begin(), this->A.end(), \
68                             [&el](std::pair<T, int> &i) {
69                                 return el == i.first;
70                             }
71     );
72
73     if(it != this->A.end())
74         return std::distance(this->A.begin(), it);
75     else
76         return -1;
77 }

```

Codice 2.19: minHeap.hpp

```

1  #ifndef _MINHEAP_HPP_
2  #define _MINHEAP_HPP_
3
4  #include <priorityqueue/heap/heap.hpp>
5
6  /** MinHeap using all attributes and methods from Heap */
7  template <class T>
8  class MinHeap : public Heap<T> {
9      public:

```

```

10      /** build a MinHeap from elements inserted in A[n] in O(n) */
11      void buildHeap();
12
13      /** Min heapify procedure makes magic, threatening A[n] as a BT.
14       * O(log n) */
15      void heapify(int i);
16
17      /** creasily (Heap)sorting A[n] in O(n*logn) */
18      void heapSort();
19  };
20
21  #endif //_MINHEAP_HPP_

```

Codice 2.20: minHeap.cpp

```

1  #include <priorityqueue/heap/minHeap.hpp>
2  #include "heap.cpp"
3
4  template <class T>
5  void MinHeap<T>::heapify(int i){
6      if( i >= 0 ){
7          int min = 0;
8          int l = Heap<T>::left(i);
9          int r = Heap<T>::right(i);
10
11          //check if l exist and then look for a min
12          if( l <= this->getHeapSize()-1 && \
13              this->A.at(i).second > this->A.at(l).second )
14              min = l;
15          else
16              min = i;
17
18          // if not l neither i is min t then r is
19          if( r <= this->getHeapSize()-1 && \
20              this->A.at(min).second > this->A.at(r).second )
21              min = r;
22
23          //if max is not i then swap and re-run heapify recursively
24          if(min != i){
25              this->swap(i, min);
26              this->heapify(min);
27          }
28      }
29  }
30
31
32  template <class T>

```

```

33 void MinHeap<T>::buildHeap(){
34     // set HeapSize at first run
35     this->setHeapSize(this->A.size());
36
37     // run heapify starting from middle going up
38     for(auto j = (this->getHeapSize()/2); j >= 0; j--){
39         this->heapify(j);
40     }
41
42
43     template <class T>
44     void MinHeap<T>::heapSort(){
45         // build a Min heap at first
46         this->buildHeap();
47
48         for(auto j = this->A.size()-1; j > 0; j--){
49             // Since A[0] is the greatest put it in last pos
50             this->swap(0, j);
51             this->decrHeapSize();
52             this->heapify(0);
53         }
54     }

```

Codice 2.21: minPriorityQueue.hpp

```

1  #ifndef _MINPRIORITYQUEUE_HPP_
2  #define _MINPRIORITYQUEUE_HPP_
3
4  #include <priorityqueue/heap/minHeap.hpp>
5
6  /**
7   * min priority queue created using a Min Heap
8   */
9  template <class T>
10 class MinPriorityQueue {
11     private:
12         MinHeap<T> A; // used for operations
13
14     public:
15         MinPriorityQueue(){};
16         ~MinPriorityQueue(){};
17
18         /** Insert an element with his key in O(log n) */
19         void insert(T el, int key);
20
21         /** get minimum in O(1) */
22         std::pair<T, int> minimum();

```

```

23
24     /** Extract the minimum in  $O(\log n)$  */
25     std::pair<T, int> extractMin();
26
27     /** decrease the priority of an element by a key value */
28     void decreasePriority(T el, int key);
29
30     /** Printing the priority queue */
31     void print(){this->A.printHeap();};
32     void printAll(){this->A.printA();};
33
34     /** return size of priority queue */
35     int getPQSize(){return A.getHeapSize();};
36 };
37
38 #endif // _MINPRIORITYQUEUE_HPP_

```

Codice 2.22: minPriorityQueue.cpp

```

1  #include <priorityqueue/minPriorityQueue.hpp>
2  #include <other/debug.hpp>
3
4  #include "heap/minHeap.cpp"
5
6  #include <cassert>
7
8  // insert new T el and decrease his priority for
9  // correct positioning
10 template<class T>
11 void MinPriorityQueue<T>::insert(T el, int priority){
12     this->A.insert(el, priority);
13     this->A.incrHeapSize();
14     decreasePriority(el, priority);
15 }
16
17
18 // return minimum in queue
19 template <class T>
20 std::pair<T, int> MinPriorityQueue<T>::minimum(){
21     return this->A.at(0);
22 }
23
24
25 // extract minimum
26 template <class T>
27 std::pair<T, int> MinPriorityQueue<T>::extractMin(){
28     std::pair<T, int> min;

```

```

29     if(this->A.getHeapSize() < 1)
30         return min;
31
32     min = this->A.at(0);
33     this->A.swap(0, this->A.getHeapSize()-1);
34     this->A.decrHeapSize();
35     // no need to delete if there are no new element inserted
36     // this->A.popBack();
37
38     // restore property
39     this->A.heapify(0);
40
41     return min;
42 }
43
44
45
46 template <class T>
47 void MinPriorityQueue<T>::decreasePriority(T el, int priority){
48     int i = this->A.find(el);
49     if(i == -1)
50         return;
51
52     if(priority > this->A.at(i).second){
53         IFDEBUG debug_print << "no need to decrease" << std::endl;
54         return;
55     }
56
57     // set new priority
58     this->A.set(i, std::make_pair(A.at(i).first, priority));
59
60     // move down untill father is lower
61     while( i > 0 && this->A.at(this->A.parent(i)).second >
62           ↪ this->A.at(i).second ) {
63         this->A.swap(this->A.parent(i), i);
64         i = this->A.parent(i);
65     }
66 }

```

Codice 2.23: debug.hpp

```

1  #ifndef _DEBUG_HPP_
2  #define _DEBUG_HPP_
3
4  // debugging
5  #include <iostream>
6  #define debug_print std::cerr

```



```

7
8     #ifdef DEBUG
9         #define IFDEBUG if(1)
10    #else
11        #define IFDEBUG if(0)
12    #endif
13
14 #endif // _DEBUG_HPP

```

Codice 2.24: item.hpp

```

1 #ifndef _ITEM_HPP_
2 #define _ITEM_HPP_
3
4 #include <limits>
5
6 /**
7  * @brief Generic Item
8  * @details using a double as a key
9  * @tparam Data parameter as value
10  */
11 template <class Data>
12 class Item {
13     private:
14         double _key;
15         Data _data;
16
17     public:
18         //Item() : _key(std::numeric_limits<int>::min()), _data {} {}
19         Item(double k) : _key(k), _data {} {}
20         Item(double k, Data &d) : _key(k), _data(d) {}
21
22         //generic setter and getter
23         void setKey(double k) { this->_key = k; }
24         double getKey() { return this->_key; }
25
26         void setData(Data &data) { this->_data = data; }
27         Data& getData() { return this->_data; }
28 };
29
30 #endif // _ITEM_HPP_

```

Codice 2.25: vertex.hpp

```

1 #ifndef _VERTEX_HPP_
2 #define _VERTEX_HPP_

```

```

3
4 #include <limits>
5 #include <utility> //make_pair()
6 #include <unordered_map>
7
8 #include <other/debug.hpp>
9 #include <other/item.hpp>
10
11 // dijkstra algorithm
12 #define inf std::numeric_limits<int>::max()
13
14 // weighted node
15 #define WeightedVertex std::pair< Vertex *, int >
16
17 // using adjacent map instead of vector -> O(1) direct access
18 #define AdjacentMap std::unordered_map< Vertex *, int >
19
20
21 /**
22  * @brief Vertex class inherits from generic Item
23  * with double as key and void* as data.
24  */
25 class Vertex : public Item<void *>{
26 private:
27     double _d = inf; // distance
28     Vertex *_prev = nullptr; // parent
29     AdjacentMap _adjmap; // map
30
31 public:
32     Vertex(int key) : Item<void *>(key) {}
33     ~Vertex(){}
34
35     void setParent(Vertex *p) { this->_prev = p; }
36     Vertex* getParent() { return this->_prev;}
37
38     void setD(double d) { this->_d = d; }
39     double getD() { return this->_d; }
40
41     // add edge to a node in map
42     void addEdge(WeightedVertex u);
43
44     void printAdj();
45     AdjacentMap getAdj() { return this->_adjmap; };
46
47     // overload of operator for comparing situation
48     bool operator== (Vertex &other) {return this->getKey() ==
49         ⇨ other.getKey();}
50     bool operator!= (Vertex &other) {return this->getKey() !=
51         ⇨ other.getKey();}

```

```

50
51     // printing purpose
52     friend std::ostream& operator<<(std::ostream &Str, Vertex *v) {
53         Str<<v->getKey()+1<<"\n";
54         return Str;
55     };
56 };
57
58 #endif // _VERTEX_HPP_

```

Codice 2.26: vertex.cpp

```

1  #include <graph/vertex.hpp>
2
3  // insert node and weight
4  void Vertex::addEdge(WeightedVertex u) {
5      this->_adjmap.insert(u);
6  }
7
8
9  void Vertex::printAdj() {
10     std::cout << "Adj list of [" << this->getKey()+1 <<"]:"<< std::endl;
11     for(auto it = this->_adjmap.begin(); it != this->_adjmap.end(); it++) {
12         std::cout << " " << it->second << " to reach --> [";
13         std::cout << it->first->getKey()+1 <<"]"<< std::endl;
14     }
15 }

```

Codice 2.27: graph.hpp

```

1  #ifndef _GRAPH_HPP_
2  #define _GRAPH_HPP_
3
4  #include <list>
5  #include <vector>
6  #include <functional>
7
8  #include <graph/vertex.hpp>
9  #include <priorityqueue/minPriorityQueue.hpp>
10
11 #define Vertices std::vector<Vertex *>
12
13 /**
14  * @brief Undirected Graph class

```

```

15  *
16  */
17  class Graph {
18  private:
19      const int _V; // no need to modify once set
20      Vertices *_vertices;
21
22  public:
23      Graph(int vertices);
24      ~Graph(){ delete [] _vertices; }
25
26      // return all vertex
27      Vertices* getVertices() {return _vertices;}
28
29      //get single vertex if correctly indexed
30      Vertex* getVertex(int i) { return (i < (int)this->_vertices->size() ?
31      ↪ this->_vertices->at(i) : nullptr) ;}
32
33      void printGraph();
34
35      // add two Edge per call (undirected)
36      void addEdge(Vertex *a, Vertex *b, int weight);
37
38      /* Apply dijkstra algorithm starting from a node */
39      void dijkstra(Vertex *s);
40
41      /* overloaded dijkstra -> get a lambda function for stop criteria
42      ↪ after
43      * extracted node finished relaxing. Return extracted node. */
44      Vertex* dijkstra(Vertex *s, std::function <bool (Vertex *)>const&
45      ↪ lambda);
46
47      /* return a path from node D up to last prev (nullptr) or S */
48      std::vector<Vertex> getPath(Vertex *s, Vertex *d);
49
50      /* print path */
51      int printPath(Vertex *s, Vertex *d, bool print = false, bool reverse =
52      ↪ false);
53      int printPath(Vertex *s, Vertex *d, std::vector<Vertex> &path, bool
54      ↪ print = false, bool reverse = false);
55  };
56
57  #endif // _GRAPH_HPP_

```

Codice 2.28: graph.cpp

```

1  #include <graph/graph.hpp>
2  #include <cassert>

```

```

3
4 #include "../priorityqueue/minPriorityQueue.cpp"
5
6 /* default constructor, allocating vertices from 0 to _V */
7 Graph::Graph(int vertices) : _V(vertices) {
8     this->_vertices = new Vertices(_V);
9
10    int i = 0;
11    for(auto v = this->_vertices->begin(); v != this->_vertices->end(); v++)
12        (*v) = new Vertex(i++); // i as key
13 }
14
15
16 void initSingleSource(Vertices *v, Vertex *source) {
17     for(auto vv = v->begin(); vv != v->end(); vv++ ) {
18         (*vv)->setD(inf); // set max int as D
19         (*vv)->setParent(nullptr); // no parent discovered
20     } // init all vertices distance and parents
21     source->setD(0); // exept source
22 }
23
24
25 void relax(Vertex *u, Vertex *v, int w, MinPriorityQueue<Vertex *> *Q) {
26     if(u->getD() == inf && u->getParent() == nullptr)
27         u->setD(0); // fix for not connected graph that has D still as inf
28
29     if(v->getD() > u->getD() + w) { // if adj node Dist is bigger then relax
30         v->setD(u->getD() + w); // set to lower
31         v->setParent(u); // parent found
32         Q->decreasePriority(v, v->getD()); // set priority low
33     }
34 }
35
36
37 void Graph::dijkstra(Vertex *s) {
38     //init all source and creaty a Minimum priority queue
39     initSingleSource(this->_vertices, s);
40     MinPriorityQueue<Vertex *> *Q = new MinPriorityQueue<Vertex *>();
41
42     for(auto v = this->_vertices->begin(); v != this->_vertices->end(); v++
43         ↪ )
44         Q->insert((*v), (*v)->getD()); // insert all vertices
45
46     while( Q->getPQSize() != 0 ) {
47         Vertex *u = Q->extractMin().first;
48         auto uAdj = u->getAdj();
49         for(auto v : uAdj)
50             relax(u, v.first, v.second, Q);
51     }

```

```

51 }
52
53
54 Vertex* Graph::dijkstra(Vertex *s, std::function <bool (Vertex *)>const&
    ↪ lambda) {
55     //init all source and creaty a Minimum priority queue
56     Vertex * u = nullptr;
57     initSingleSource(this->_vertices, s);
58     MinPriorityQueue<Vertex *> *Q = new MinPriorityQueue<Vertex *>();
59
60     for(auto v = this->_vertices->begin(); v != this->_vertices->end(); v++)
        ↪ )
61         Q->insert((*v), (*v)->getD()); // insert all vertices
62
63     while( Q->getPQSize() != 0 ) {
64         u = Q->extractMin().first;
65         auto uAdj = u->getAdj();
66         for(auto v : uAdj)
67             relax(u, v.first, v.second, Q);
68
69         // if lambda func return true, stop dijkstra
70         if(lambda(u))
71             break;
72     }
73     // and return lastest extracted node
74     return u;
75 }
76
77
78 std::vector<Vertex> Graph::getPath(Vertex *s, Vertex *d) {
79     auto tmp = d;
80     std::vector<Vertex> path;
81
82     // tmp goes up untill no more nodes in graph
83     // or source found
84     while(tmp->getParent() != nullptr && tmp != s) {
85         path.push_back(*tmp);
86         tmp = tmp->getParent();
87     }
88
89     // if previous while stopped, insert source in path
90     if(tmp->getParent() != nullptr)
91         tmp = tmp->getParent();
92
93     path.push_back(*tmp);
94
95     return path;
96 }
97

```

```

98
99 int Graph::printPath(Vertex *s, Vertex *d, std::vector<Vertex> &path, bool
   ↪ print, bool reverse) {
100     auto sum = 0;
101
102     // if path hasn't source and dest as first and last, don't print
103     if(path.back() != *s || path.front() != *d) {
104         if(print)
105             std::cout << "there are no paths that connect " << s->getKey()+1
106             << " with " << d->getKey()+1 << std::endl;
107         sum = -1;
108     }
109     else {
110         if(!reverse) // inserted node from dest to source,
111             std::reverse(path.begin(), path.end()); // need to reverse
112
113         if(print)
114             std::cout << path.at(0).getKey()+1;
115
116         for(auto i = 1; i < (int)path.size(); i++ ) {
117             if(print)
118                 std::cout << "-"<< path.at(i).getKey()+1;
119
120             if(reverse)
121                 sum += path.at(i-1).getD() - path.at(i).getD();
122             else
123                 sum += path.at(i).getD() - path.at(i-1).getD();
124         }
125     }
126     return sum; // O(E) print and sum path weight returned
127 }
128
129
130 int Graph::printPath(Vertex *s, Vertex *d, bool print, bool reverse) {
131     // if no path given, calculate it
132     auto path = getPath(s, d);
133     return printPath(s, d, path, print, reverse);
134 }
135
136
137 // add undirected edge
138 void Graph::addEdge(Vertex *a, Vertex *b, int weight) {
139     a->addEdge(WeightedVertex(b, weight));
140     b->addEdge(WeightedVertex(a, weight));
141 }
142
143 // for all nodes, print adj map
144 void Graph::printGraph() {
145     for(auto x = this->_vertices->begin(); x != this->_vertices->end(); x++)

```

```
146      (*x)->printAdj();  
147  }
```

Codice 2.29: galacticgraph.hpp

```

1  #ifndef _GALACTICTGRAPH_HPP_
2  #define _GALACTICTGRAPH_HPP_
3
4  #include <unordered_map>
5  #include <graph/graph.hpp>
6
7  /**
8    * @brief GalacticGraph. Using wormholes for fast travel.
9    * if there is a wormhole near s and d,
10   * use that to teleport in 1 unit time.
11   */
12  class GalacticGraph : public Graph {
13  private:
14      std::unordered_map<Vertex *, int> _wormholes;
15      bool isWormhole(Vertex *x); // return true if a node is in hashmap
16
17  public:
18      GalacticGraph(int V) : Graph(V) {};
19
20      // compute path from s to d, then from s to first worm and reapply
21      ↪ dijkstra
22      // using d as source, and get path from d to worm
23      void galacticDijkstra(Vertex *s, Vertex *d);
24
25      // load a vertex in hashmap
26      void addWormhole(Vertex *w){this->_wormholes.insert(std::make_pair(w,
27          ↪ 1));};
28  };
29
30  #endif // _GALACTICTGRAPH_HPP_

```

Codice 2.30: galacticgraph.cpp

```

1  #include <galacticgraph.hpp>
2
3
4  bool GalacticGraph::isWormhole(Vertex *x) {
5      if(x != nullptr)
6          return this->_wormholes.find(x) != this->_wormholes.end();
7      // O(1) search
8      return false;
9  }

```

```

10
11
12 void GalacticGraph::galacticDijkstra(Vertex *s, Vertex *d) {
13     auto wFound = new Vertices();
14
15     // calculate minimum path tree from source
16     // saving all wormhole found
17     dijkstra(s, [this, d, wFound](Vertex *tmp){
18         if(isWormhole(tmp))
19             wFound->push_back(tmp);
20         return tmp == d; // stop criteria
21     });
22
23     std::cout<<"Travel without wormholes: ";
24     auto traveldist = printPath(s, d, true);
25     if(traveldist>0)
26         std::cout<<" in "<< traveldist <<" time unit\n\n";
27     else
28         std::cout<<"\t(disconnected graph)\n\n";
29
30     auto w1 = wFound->empty() ? nullptr : wFound->front();
31     // if there was a wormhole in path, get the first extracted
32     if(w1 != nullptr) {
33         auto p1 = getPath(s, w1); // calculate path
34
35         wFound->clear();
36         // calculate minimum path tree from source
37         dijkstra(d, [this, s, wFound](Vertex *tmp){
38             if(isWormhole(tmp))
39                 wFound->push_back(tmp);
40             return tmp == s; // stop criteria
41         });
42
43         // if there was a wormhole in path, get the first one extracted
44         auto w2 = wFound->empty() ? nullptr : wFound->front();
45
46         if(w2 != nullptr && w1 != w2) { // if wormhole are different
47             std::cout<<"Using wormholes -> [" << w1->getKey()+1
48                 << ", "<< w2->getKey()+1 <<"] ";
49
50             auto t1 = printPath(s, w1, p1, true); // print first path
51             std::cout<<"^";
52             auto t2 = printPath(d, w2, true, true); // print second one half
53
54             std::cout<<" in "<<t1+1+t2<<" unit time\n\n";
55         }
56         else
57             std::cerr<<"No fast travel with wormhole.\n";
58     }

```

```

59     else
60         std::cerr<<"No wormhole from Source to Destination.\n";
61 }

```

Codice 2.31: loader.hpp

```

1  #ifndef _LOADER_HPP_
2  #define _LOADER_HPP_
3
4  #include <string>
5  #include <galacticgraph.hpp>
6
7  /**
8   * @brief Loader class for parse file.txt into Graph
9   */
10 class Loader {
11     private:
12         std::string _filename;
13         GalacticGraph *_G = nullptr;
14
15     public:
16         Loader(const std::string fn);
17         ~Loader(){ delete _G;}
18
19         GalacticGraph *getGraph(){return this->_G;}
20 };
21
22 #endif //_LOADER_HPP_

```

Codice 2.32: loader.cpp

```

1  #include <iostream>
2  #include <fstream>
3
4  #include <loader.hpp>
5
6  Loader::Loader(const std::string fn) : _filename(fn) {
7
8      std::ifstream filestream(this->_filename);
9      if(filestream.fail()){
10         std::cerr << "Error opening file" << std::endl;
11         exit(EXIT_FAILURE);
12     }
13 }

```

```

14     int V, E, W;
15     filestream >> V;
16     filestream >> E;
17     filestream >> W;
18
19     if(V < 1 || V > 1000 ) {
20         std::cerr << "Solar systems must be at least 2 and less then 1k " <<
21             ↪ std::endl;
22         exit(EXIT_FAILURE);
23     }
24
25     if(E < 0 || E > 10000 ) {
26         std::cerr << "Connections must be at least one and less then 10k" <<
27             ↪ std::endl;
28         exit(EXIT_FAILURE);
29     }
30
31     if(W < 1 || W > V) {
32         std::cerr << "Wormholes must be at least 2 and less then V("<<V<<")"
33             ↪ << std::endl;
34         exit(EXIT_FAILURE);
35     }
36
37     std::cout << "Filling a graph with "<< V<< " nodes, ";
38     std::cout << E << " edges, ";
39     std::cout << W << " wormholes\n";
40
41     this->_G = new GalacticGraph(V);
42     auto vertices = this->_G->getVertices();
43
44     IFDEBUG debug_print << "STARTING READING FILE: " <<E-W<< std::endl;
45     for(auto it = 0; it < E-W; it++) {
46         int a, b, w;
47         filestream >> a;
48         filestream >> b;
49         filestream >> w;
50         this->_G->addEdge(vertices->at(a-1), vertices->at(b-1), w);
51     }
52
53     IFDEBUG debug_print << "STARTING READING 2ndhalf: "<<W<< std::endl;
54     for(auto it = 0; it < W; it++) {
55         int a, b, w;
56         filestream >> a;
57         filestream >> b;
58         filestream >> w;
59         this->_G->addEdge(vertices->at(a-1), vertices->at(b-1), w);
60         this->_G->addWormhole(vertices->at(b-1));
61     }

```

```
60
61     filestream.close();
62     return;
63 }
```

Bibliografia

- [1] *Introduzione agli algoritmi e strutture dati (Italiano)*, 2010
Thomas H. Cormen (Autore), Charles E. Leiserson (Autore), Ronald L. Rivest (Autore), Clifford Stein (Autore), L. Colussi (a cura di).
- [2] Slide del corso *Algoritmi e Strutture Dati e Laboratorio di Algoritmi e Strutture Dati*, tenuto dai docenti *F. Camastra* e *A. Ferone*.
- [3] Overleaf Documentation
<https://www.overleaf.com/learn/>
- [4] L'arte del L^AT_EX, *Lorenzo Pantieri*
<http://www.lorenzopantieri.net/LaTeX.html>
- [5] L^AT_EX/Source Code Listings
https://en.wikibooks.org/wiki/LaTeX/Source_Code_Listings
- [6] Curiously recurring template pattern
https://en.wikipedia.org/wiki/Curiously_recurring_template_pattern
- [7] Curiously Recurring Template Pattern
<https://www.fluentcpp.com/2017/05/12/curiously-recurring-template-pattern/>