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

Elenco delle figure

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 binario di ricerca bilanciato, **albero rosso-nero** o **albero red-black**, e delle **hash table**: 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, il quale puntando ad una hash table utilizza la chiave 2 per associare la stringa. 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: $\forall\ x \in \text{BST}:\ key(x.left) \leq key(x) < key(x.right)$. 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 **logarimtica** data dalla profondità dell'albero. Il problema sorge nel caso in cui avvengono cancellazioni sbagliate o inserimen-

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

ti sbagliati che seppur mantengono la proprietà degli ABR, degradano tale albero in una lista concatenata compromettendo 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 garantire 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. Precisamente ad ogni chiave va applicata una funzione detta funzione di hashing che calcolerà un indice. Può capitare però che una funzione hash applicata su chiavi diverse indicizzi celle simili dell'hashtable, per tanto bisogna gestire queste *collisioni*. La tavola hash realizzata 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 *chiave*1 : *chiave*2 : *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:

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

 1. Insert
 2. Remove
 3. Query
 4. Print
 0. Exit
>_
```

Tutte le opzioni restituiscono un output di avvenuta operazione con dettagli su eventuali errori, ad eccezzione 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'insertimento nella struttura dati creata va effettuare prima una ricerca del nodo di chiave *key*1. 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 *key*2, se anche questa ricerca restituisce un esito negativo allora si procede con l'inserimento.

Algorithm: Insert

```
   input : key1, key2, string
   result: boolean
 1 node = Search in Red Black with key1;
 2 if ∄ node :
 3 |   new Hashtable();
 4 |   new Node();
 5 |   Hashtable.insert(key2, string);
 6 |   Node.insert(key1, Hashtable);
 7 |   return true;
 8 else:
 9 |   if ∄ 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.

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 ∃ 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 risconto 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 ≠ 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 templato 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 base template (e.g. AbstractBinaryNode), 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.

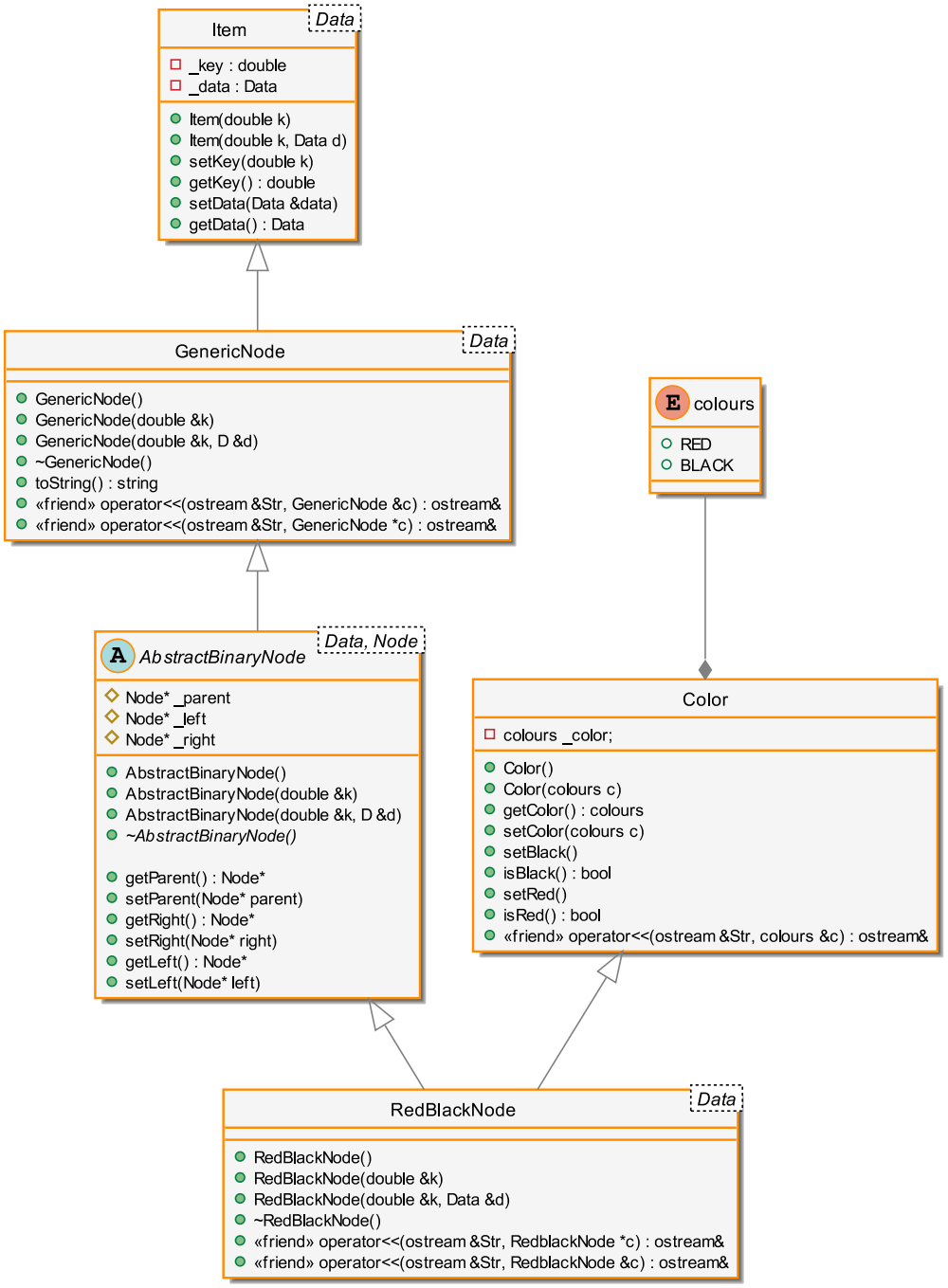

Figura 1.1: Nodi degli alberi

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* richiama tramite lo scope della classe BinarySearchTree la insert e successivamente applica il **fix** tipico dei red black.

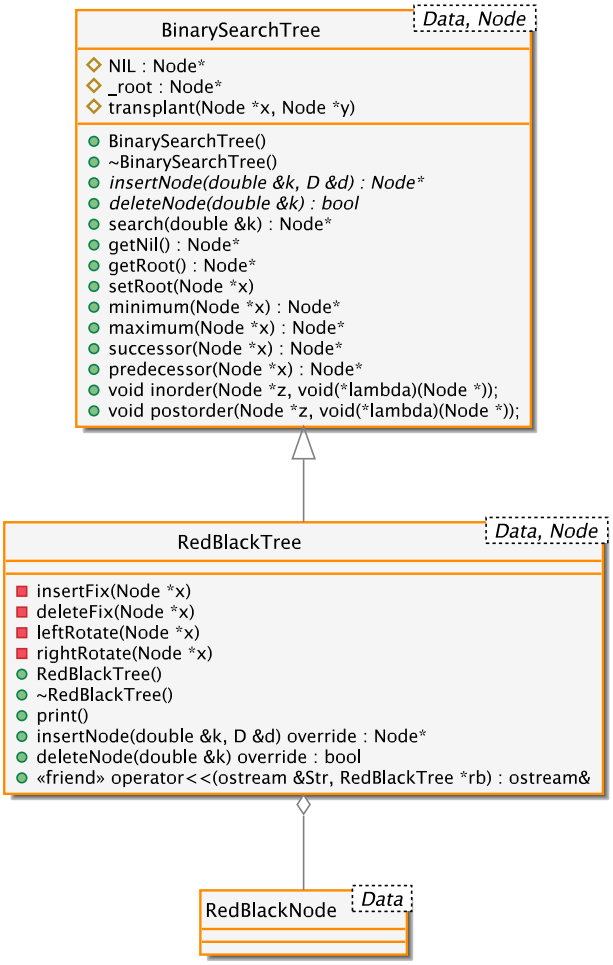

Figura 1.2: Alberi

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 templato, il dato da conservare e la funzione di hash sotto forma di classe. 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 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 $h(i, m) = (h_1(k) + i * h_2(k))\ mod\ m$. Sebbene con un costo computazionale più alto, il doppio hashing risolve le collisioni più in fretta rispetto allispezione lineare o quadratica.

La funzione *search*() nella class **HashTable** usa due diversi parametri di input

[2] m è il size dell'hashtable

per restituire valori diversi. Con la ricerca per valore si restituise, se disponibile un indice che usato nella seconda funzione di ricerca restituisce il dato.

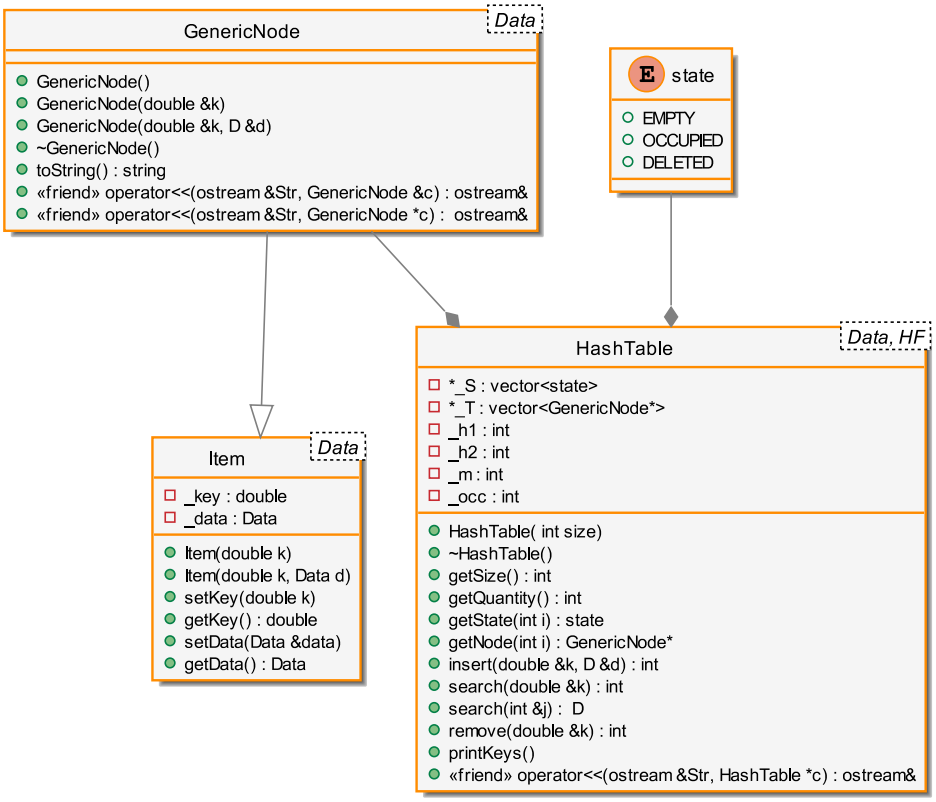

Figura 1.3: Hash Table

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 $O(n)$, 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) $\alpha \to 1$.

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 $\mathbf{O(log_2 n)}$ e $\mathbf{O(m + log_2 n)}$.;
- 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 $\mathbf{O(log_2 n)}$.

L'operazione di **ricerca** impiega $\mathbf{O(m + log_2 n)}$ 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 $\mathbf{O(log_2 n)}$.
La **cancellazione** ha complessità simili all'inserimento: 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 $\mathbf{O(log_2 n)}$ oppure $\mathbf{O(m + log_2 n)}$.

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 stai 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.

```
123:321:hello
2:4:54
1:4:test
2:4:h&:o
2:12:asd
```

Listing 1.1: File di prova con chiavi duplicate

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 mostrare come l'albero red black sia un albero binario.

```
$ ./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

```
#ifndef _DEBUG_HPP_
#define _DEBUG_HPP_

#include <iostream>

/**
 * printer and debugging purpouse
 */

#define debug_print std::cerr

#ifdef DEBUG
  #define IFDEBUG if(1)
#else
  #define IFDEBUG if(0)
#endif

#endif //_DEBUG_HPP_
```

```
#ifndef _ITEM_HPP_
#define _ITEM_HPP_

#include <limits>

/**
 * @brief Generic Item
 * @details using a double as a key
 * @tparam Data parameter as value
 */
template <class Data>
class Item {
  private:
    double _key;
    Data _data;

  public:
    Item() : _key(std::numeric_limits<double>::min()), _data {} {}
    Item(double &k) : _key(k), _data {} {}
    Item(double &k, Data &d) : _key(k), _data(d) {}

    //generic setter and getter
    void setKey(double &k) { this->_key = k; }
    double& getKey() { return this->_key; }

```

```
26      void setData(Data &data) { this->_data = data; }
27      Data& getData() { return this->_data; }
28  };
29
30  #endif //_ITEM_HPP_
```

```cpp
#ifndef _GENERICNODE_HPP_
#define _GENERICNODE_HPP_

#include <string>
#include <iostream>

#include <other/item.hpp>

/**
 * @brief Generic Node class
 * @tparam N template base class
 */
template <typename D>
class GenericNode : public Item<D> {
  public:
    GenericNode() : Item<D>() {};
    GenericNode(double &k) : Item<D>(k) {};
    GenericNode(double &k, D &d) : Item<D>(k, d){};
    ~GenericNode() {};
    std::string toString(){return
     ↪  "["+std::to_string((int)this->getKey())+"] : "+
      std::string(this->getData())+"\n";}

    friend std::ostream& operator<<(std::ostream &Str, GenericNode<D> &c){
      Str<<c.toString(); return Str; }

    friend std::ostream& operator<<(std::ostream &Str, GenericNode<D> *c){
      Str<<c->toString(); return Str; }
};

#endif //_GENERICNODE_HPP_
```

```cpp
#ifndef _IBNODE_HPP_
#define _IBNODE_HPP_

#include <iostream>

#include <nodes/genericnode.hpp>

/**
 * @brief AbstractBinaryNode
 * @details Using CRTP as base (Used in RBNode for example)
 * https://en.wikipedia.org/wiki/Curiously_recurring_template_pattern
 * @tparam D data template for value to store in node
 * @tparam N template for CRTP pattern
 */
template <typename D, class Node>
class AbstractBinaryNode : public GenericNode<D> {
  protected:
    // since is a Binary Node we got a parent and only 2 child
    Node* _parent = nullptr; // defaults values
    Node* _left = nullptr;
    Node* _right = nullptr;

  public:
    AbstractBinaryNode() : GenericNode<D>() {};
    AbstractBinaryNode(double &k) : GenericNode<D>(k) {};
    AbstractBinaryNode(double &k, D &d) : GenericNode<D>(k, d) {};
    virtual ~AbstractBinaryNode() {};

    //various setter and getter
    Node* getParent(){ return this->_parent; };
    void setParent(Node* parent){ this->_parent=parent; };

    Node* getRight(){ return this->_right; };
    void setRight(Node* right){ this->_right=right; };

    Node* getLeft(){ return this->_left; };
    void setLeft(Node* left){ this->_left=left; };
};

#endif //_IAbstractBNode_HPP_
```

```
#ifndef _COLOR_HPP_
#define _COLOR_HPP_

#include <iostream>

/**
 * @brief Colours tag as enum
 */
enum colours {RED, BLACK};


/**
 * @brief Color setter of a node
 * @details used for generic implementation
 */
class Color {
  private:
    colours _color;

  public:
    Color();
    Color(colours c) : _color(c) {};

    //various setter and getter
    colours getColor(){return this->_color;}
    void setColor(colours c){this->_color = c;}

    void setBlack(){this->_color = BLACK;}
    bool isBlack(){return (this->_color == BLACK);}

    void setRed(){this->_color = RED;}
    bool isRed(){return (this->_color == RED);}

    friend std::ostream& operator<<(std::ostream &Str, colours &c){
      (c==RED) ? Str<<"red" : Str<<"black";
      return Str;
    }
};

#endif //_COLOR_HPP_
```

```
#ifndef _REDBLACKNODE_HPP_
#define _REDBLACKNODE_HPP_

#include <nodes/abstractbinarynode.hpp>
#include <nodes/color.hpp>

/**
 * @brief Red Black Node
 * @details Inherites from Binary Node and using
 * CRTP for a dynamic inheritance. Also extending Color
 * @tparam D
 */
template <typename D>
class RedBlackNode : public AbstractBinaryNode<D, RedBlackNode<D>>, public
 ↪  Color {
  public:
    RedBlackNode() : Color(BLACK) {};
    RedBlackNode(double &k) : AbstractBinaryNode<D, RedBlackNode<D>>(k),
     ↪  Color(BLACK) {};
    RedBlackNode(double &k, D &d) : AbstractBinaryNode<D,
     ↪  RedBlackNode<D>>(k, d), Color(BLACK) {};
    ~RedBlackNode() {
      if(this->_left==nullptr)
        delete this->_left;
      if(this->_right==nullptr)
        delete this->_right;
      if(this->_parent==nullptr)
        delete this->_parent;
    };

    friend std::ostream& operator<<(std::ostream &Str, RedBlackNode<D> &c)
     ↪  {
      if(c.isRed())
        Str<<"\033[31m";
      Str<<"("<<c.getKey()<<") : \n"<<c.getData()<<"\033[0m";
      return Str;
    }

    friend std::ostream& operator<<(std::ostream &Str, RedBlackNode<D> *c)
     ↪  {
      if(c->isRed())
        Str<<"\033[31m";
      Str<<"("<<c->getKey()<<") : \n"<<c->getData()<<"\033[0m";
      return Str;
    }
};


```

```
44 #endif //_REDBLACKNODE_HPP_
```

```hpp
#ifndef _HASHTABLE_HPP_
#define _HASHTABLE_HPP_

#include <limits>
#include <vector>
#include <cmath>

#include <nodes/genericnode.hpp>


/*! Enum for state of single HashNode */
enum state { EMPTY, OCCUPIED, DELETED };

/**
 * @brief HashTable datastructure
 * @tparam D is the Data to store
 * @tparam HashFunction is the class that make the hashfunction
 *  to compute the index of node
 */
template <typename D>
class HashTable {
  private:
    std::vector<state> *_S; // array of states
    std::vector<GenericNode<D>*> *_T; // effective hashtable

    // using fmod -> double % int
    int _h1(double &k){return static_cast<int>(std::fmod(k, this->_m));}
    int _h2(double &k){return 1 + static_cast<int>(std::fmod(k,
     ↪  this->_m-1));}

    int _m; // default capacity
    int _occ = 0; // number of occupied nodes

  public:
    HashTable( int size = 701 );
    ~HashTable();

    /**
     * @brief Get the capacity of hashtable
     * @return int
     */
    int getSize();

    /**
     * @brief Get the current hashnode in hashtable
     * @return int
     */
    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_
```

```cpp
#include <iostream> // cerr

#include <hashtables/hashtable.hpp>


//creating GenericNode vector and state vector
template <typename D>
HashTable<D>::HashTable(int size) : _m(size) {
  this->_T = new std::vector<GenericNode<D>*>(this->_m, nullptr);
  this->_S = new std::vector<state>(this->_m, EMPTY); //init with EMPTY
   ↪  state
}


//destructing hashnodes
template <typename D>
HashTable<D>::~HashTable() {
  for(auto t = 0; t != this->_m; t++)
    if(this->_S->at(t) != EMPTY)
      delete this->_T->at(t);

  delete []_T;
  delete []_S;
}


//getter hash table size
template <typename D>
int HashTable<D>::getSize() {
  return this->_m;
}


//return the state of a indexed (j) GenericNode
template <typename D>
state HashTable<D>::getState(int i) {
  if(i >= 0 && i < this->_m)
    return this->_S->at(i);
  return DELETED;
}


// return an GenericNode indexed by i
template <typename D>
GenericNode<D>* HashTable<D>::getNode(int i) {
  if(i >= 0 && i < this->_m)
    return this->_T->at(i); // return nullptr if is not
    //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 }
```

```
1   #ifndef _BINARYSEARCHTREE_HPP_
2   #define _BINARYSEARCHTREE_HPP_
3
4
5   /**
6    * @brief Binary Search Tree
7    * @details BST inherites 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 lamba)
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_
```

```
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() :
       ↪  curr->getRight();
39   }
40
41   // set parent to previuosly saved node
42   z->setParent(prev);
43
44   // checking if is left son or right son
45   if(prev == this->NIL) // or root
46     this->_root = z;
47   else if(z->getKey() < prev->getKey())
48     prev->setLeft(z);
49   else
50     prev->setRight(z);
```

```
51
52    // new allocated node got NIL as children
53    z->setLeft(this->NIL);
54    z->setRight(this->NIL);
55
56    return z;
57  }
58
59  // delete a node
60  template<typename D, class N>
61  bool BinarySearchTree<D, N>::deleteNode(double &k) {
62    // must search
63    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   if(tmp != this->NIL) {
192     lambda(tmp); // [Catcher](Tmp){/*implementation*/}
193     postorder(tmp->getLeft(), lambda);
194     postorder(tmp->getRight(), lambda);
195   }
196 }
```

```hpp
#ifndef _REDBLACKTREE_HPP_
#define _REDBLACKTREE_HPP_

#include <trees/binarysearchtree.hpp>
#include <nodes/redblacknode.hpp>


/**
 * @brief Red Black tree
 * @details Templated Red Black tree that
 * inherites from Binary Search tree <Data, RBnode>
 * @tparam D data
 */
template <typename D, class Node = RedBlackNode<D>>
class RedBlackTree : public BinarySearchTree<D, Node>{
  private:
    // insert and delete Fix for restoring RB property
    void insertFix(Node *x);
    void deleteFix(Node *x);

    // left Rotate and right Rotate for balancing method
    void leftRotate(Node *x);
    void rightRotate(Node *x);

  public:
    RedBlackTree(){};
    ~RedBlackTree();

    //create and insert RBNode, with rb fixes
    Node* insertNode(double &k, D &d) override;

    //delete RBNode if exist
    bool deleteNode(double &k) override;

    // inorder print
    void print();

    //override for correct print
    friend std::ostream& operator<<(std::ostream &Str, RedBlackTree<D>
     ↪  *rb) {
      rb->print();
      return Str;
    };
};

#endif //_REDBLACKTREE_HPP_
```

```cpp
#include <trees/redblacktree.hpp>
#include <other/debug.hpp>

#include "binarysearchtree.cpp" // include for templated implementation workaround


// left rotate a node for balancing
template <typename D, class Node> // aka move down
void RedBlackTree<D, Node>::leftRotate(Node *x) { // O(1) op keeping BST property
  Node *y = x->getRight();  // set y
  x->setRight(y->getLeft());    // move l-subtree of y to r-subtree of x

  if ( y->getLeft() != this->NIL )
    y->getLeft()->setParent(x);

  y->setParent(x->getParent()); // link fathers

  if ( x->getParent() == this->NIL )
    this->_root = y;            // if is X was root
  else if ( x == x->getParent()->getLeft() )
    x->getParent()->setLeft(y);
  else
    x->getParent()->setRight(y);

  y->setLeft(x);
  x->setParent(y);
}


//rotate a node to right (aka move down) keeping BST property
template <typename D, class Node>
void RedBlackTree<D, Node>::rightRotate(Node *x) {
  Node *y = x->getLeft();
  x->setLeft(y->getRight());

  if ( y->getRight() != this->NIL )
    y->getRight()->setParent(x);

  y->setParent(x->getParent());

  if ( x->getParent() == this->NIL )
    this->_root = y;
  else if ( x == x->getParent()->getLeft() )
    x->getParent()->setLeft(y);
  else
    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
                ↪  same BH
72        }
73      } else {
74        y = x->getParent()->getParent()->getLeft();
75        if(y->isRed()) { // CASE 1: parent of x and uncle are RED
76          x->getParent()->setBlack();
77          y->setBlack(); // change their color and
78          x->getParent()->getParent()->setRed();
79          x = x->getParent()->getParent(); // move point to grand father
80        } else { // CASE 2: uncle is BLACK
81            if(x == x->getParent()->getLeft()) { // x is left son
82              x = x->getParent(); // move upper
83              rightRotate(x);  // then rotate so x became right son
84            } // CASE 3: // uncle is BLACK but x is left son
85            x->getParent()->setBlack();
86            x->getParent()->getParent()->setRed();
87            leftRotate(x->getParent()->getParent()); // compression aka same
                ↪  BH
88        }
89      }
90    }
91    this->_root->setBlack();
92  }
93
```

```
94
95  template <typename D, class Node>
96  Node* RedBlackTree<D, Node>::insertNode(double &k, D &d) {
97    auto z = BinarySearchTree<D, Node>::insertNode(k, d); // BST insert
98    z->setRed(); // set flag as red
99    this->insertFix(z); // fix property
100   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 
222   //  fix black root
223   //  fix two red nodes
224   //  fix Black High lose
225   if(y_original_color == BLACK)
226     deleteFix(x);
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 }
```

```hpp
#ifndef _REDBLACKHASH_HPP_
#define _REDBLACKHASH_HPP_

#include <trees/redblacktree.hpp>
#include <hashtables/hashtable.hpp>

/**
 * @brief Red Black Hash
 * @details data structure that use a Red Black to stores in every node
 * an Hashtables.
 * Key 1 is the int used to indexes a red black, Key 2 is the int to
 ↪  indexes
 * a single cell of an hashtable. D d is the data to store in hashtables
 * @tparam D data to store in hashtables
 */

template <typename D>
class RedBlackHash {
  private:
    RedBlackTree<HashTable<D> *> *_rb;
    int _size;

  public:
    //fixed size of 285700
    RedBlackHash(int s = 285700);
    ~RedBlackHash(){};

    //redblack getter
    RedBlackTree<HashTable<D> *> *getRBHashTree(){return this->_rb;};

    bool insert(int k1, int k2, D d);
    RedBlackNode<HashTable<D> *>* search(int k1, int k2, D d);
    bool remove(int k1, int k2, D d);

    // print chaining
    friend std::ostream& operator<<(std::ostream &Str, RedBlackHash<D>
     ↪  *rh) {
      Str<<rh->getRBHashTree()<<"\n";
      return Str;
    };
};

#endif //_REDBLACKHASH_HPP_
```

```hpp
#include <iostream>

```

```
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
       ↪  hashtable
25     node->getData()->insert(key2, d);               // insert string
26     return true;
27   }
28   else
29     if(node->getData()->search(key2) < 0)           // if there isnt
       ↪  already a key2
30       if(node->getData()->insert(key2, d) >= 0)     // and also there is
         ↪  space
31         return true;
32       else
33         std::cerr<<"\nerror: there is no space left in hash table in node
           ↪  "<<key1<<"\n";
34     else
35       std::cerr<<"\nerror: there is already an hashnode with key =
         ↪  "<<key2<<"\n";
36
37   // if here means that there is already a key or there is no space
38   return false;
39 }
40
41
42 template <typename D>
43 RedBlackNode<HashTable<D> *>* RedBlackHash<D>::search(int k1, int k2, D d)
   ↪  {
44   auto key1 = static_cast<double>(k1);
45   auto key2 = static_cast<double>(k2);
```

```
46
47    auto node = this->_rb->search(key1);
48
49    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<<"]
            ↪  hash["<<key2<<"]-> "<<d<<"\n"<<
56           "maybe you were looking for:
              ↪  "<<node->getData()->search(j)<<"\n";
57      }
58      else{
59        std::cerr<<"\nerror: no hash node with key2 = ["<<k2<<"]\n";
60        std::cerr<<"\nhash in node["<<key1<<"] "<<"only got keys: ";
61        node->getData()->printKeys();
62      }
63    }
64    else
65      std::cerr<<"\nerror: no RB node with key1 = ["<<k1<<"]\n";
66
67    return nullptr;
68  }
69
70
71  template <typename D>
72  bool RedBlackHash<D>::remove(int key1, int key2, D data) {
73    // if tuple is correct
74    auto node = this->search(key1, key2, data);
75    if(node != nullptr) {
76      auto k1 = static_cast<double>(key1);
77      auto k2 = static_cast<double>(key2);
78
79      // if hashtable got only one value (key1)
80      if(node->getData()->getQuantity() == 1)
81        this->_rb->deleteNode(k1); // delete RB node
82      else
83        node->getData()->remove(k2); // just remove value
84
85      return true;
86    }
87    return false;
88  }
```

```cpp
#ifndef _PARSER_HPP_
#define _PARSER_HPP_

#include <string>
#include <redblackhash.hpp>

/**
 * @brief Parser for Red Black Hashtable
 * @details Used to fill from a file an hashtable.
 *  Also providing method for C.R.D. (create, retrieve, delete) tuples
 *  tuples must be in format <int> keys and <string> data ->
 ↪  key1:key2:data
 */
class Parser {
  private:
    int _hashsize;
    std::string _filename;
    RedBlackHash<std::string> *_rh;

  public:
    Parser(const std::string fn, int hashsize = 9973);
    ~Parser(){};

    //  tuples must be in format <int> keys and <string> data ->
     ↪  key1:key2:data
    bool insert(std::string str);

    //  tuples must be in format <int> keys and <string> data ->
     ↪  key1:key2:data
    bool search(std::string str);

    //  tuples must be in format <int> keys and <string> data ->
     ↪  key1:key2:data
    bool remove(std::string str);

    //printing data structure
    void print();
};

#endif //_PARSER_HPP_
```

```cpp
#include <parser.hpp>
#include <redblackhash.hpp>
#include "redblackhash.cpp"

#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 priotià 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 ADT[3], 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[5], 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 $O(logn)$ nelle operazione di estrazione del minimo, inserimento e abbassamento priorità, poiché di facile implementazione.

[2]FIFO: first in, first out.
[3]ADT: abstract data type.
[4]Algoritmo per il calcolo dei cammini minimi da sorgente unica.
[5] Heap di Fibonacci ha complessità $O(1)$ nelle funzioni utili all' algoritmo

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 [6]

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 minino, **se esiste**: può capitare, come vedremo nel paragrafo *"Test effettuati"*, 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.

[6] ndr: 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 minimi[7] 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 womrhole 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** dal nodo sorgente a quello destinazione. 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.

[7] Anche 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 w1 = apply Dijkstra from S and save first wormhole encountered;
2 distance = printPath(s, d);
3 if ∃ w1 :
4     w2 = apply Dijkstra from D and save first wormhole encountered;
5     if ∃ w2 and w1 ≠ w2 :
6         dw1 = printPath(s, w1);
7         dw2 = printPath(w2, d);
8         d_{dw1+dw2} = dw1 + 1 + dw2;
```

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 priotità.

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.

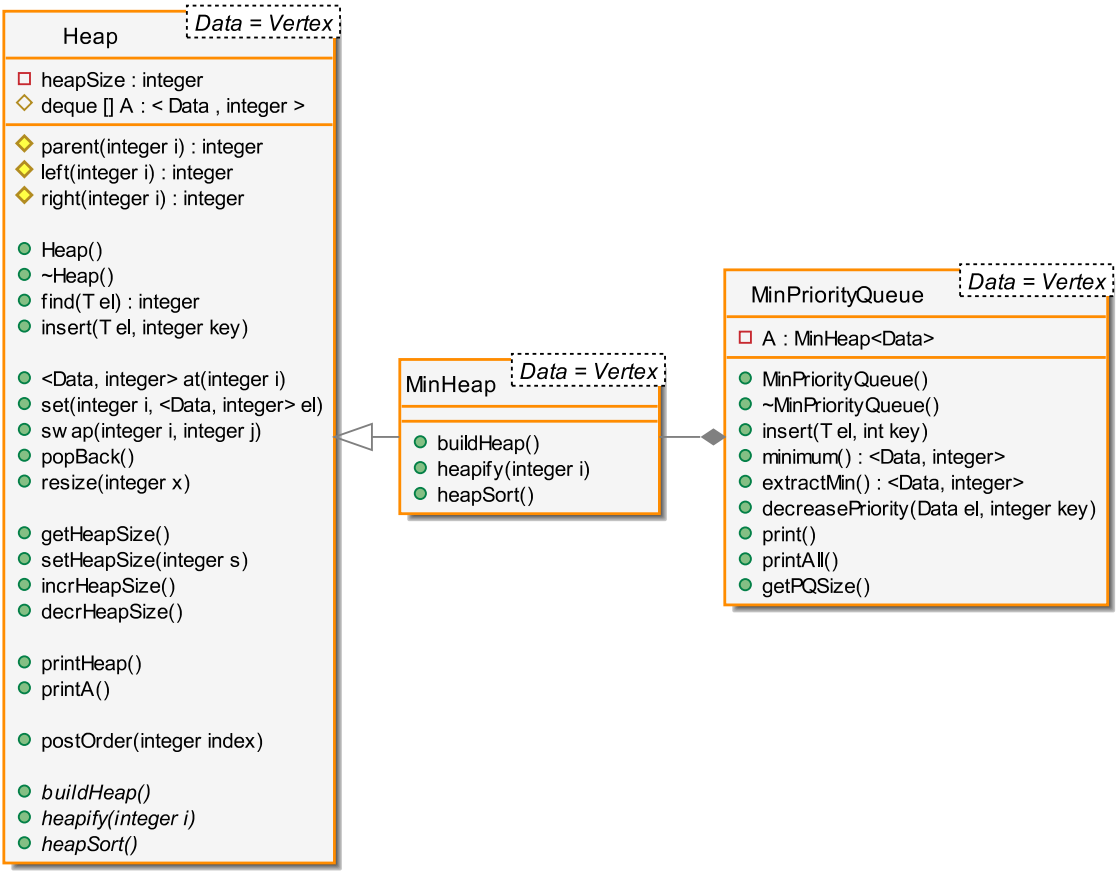

Figura 2.1: Min priority Queue

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 manuale[8] 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.

[8] Vedere riferimenti bibliografici.

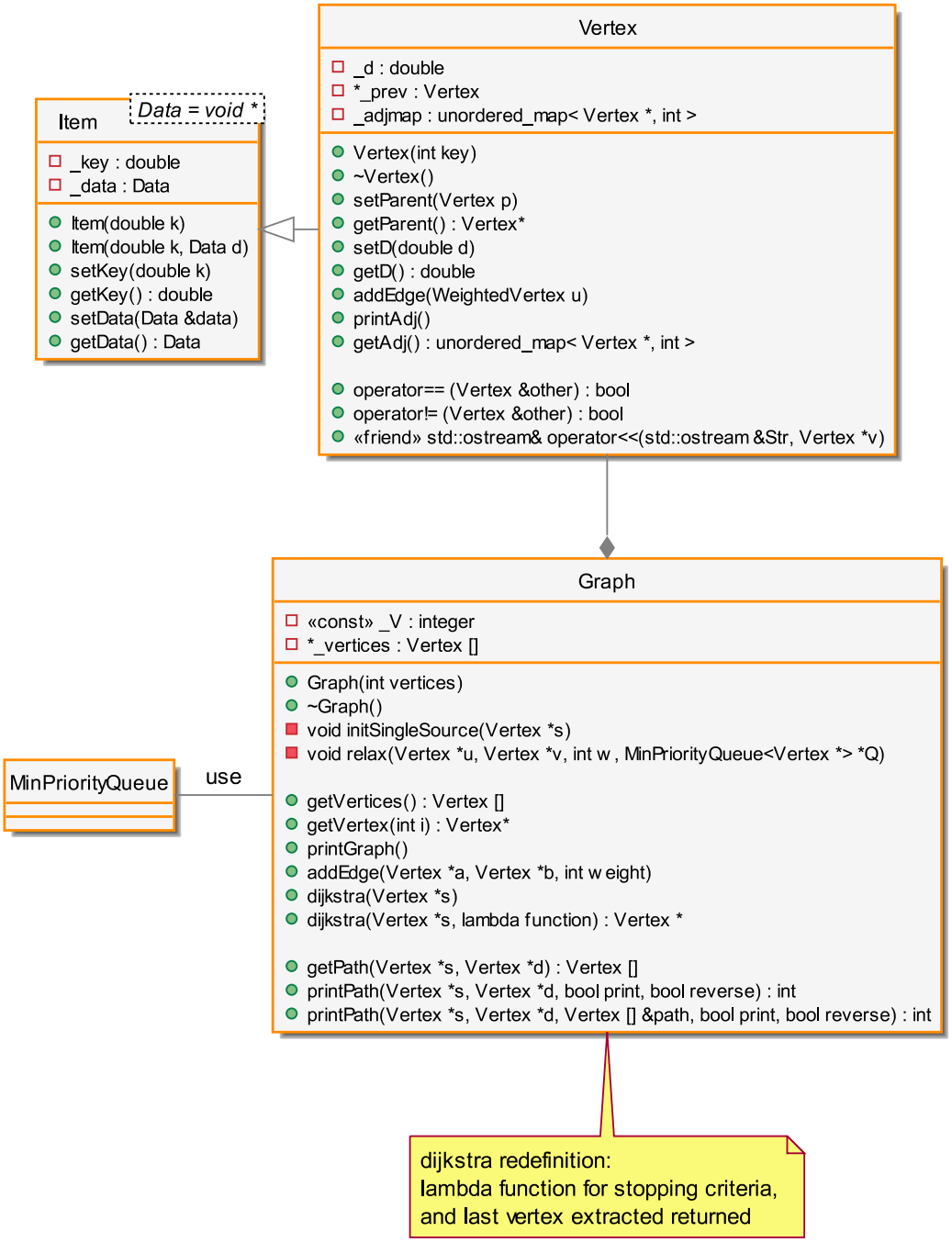

Figura 2.2: Vertici e Grafo

Parser e Grafo galattico

Il main del programma usera la classe Parser per creare e instanziare 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 sistema caricato e dell' algortimo Galactic Dijkstra (par. 2.4.1).

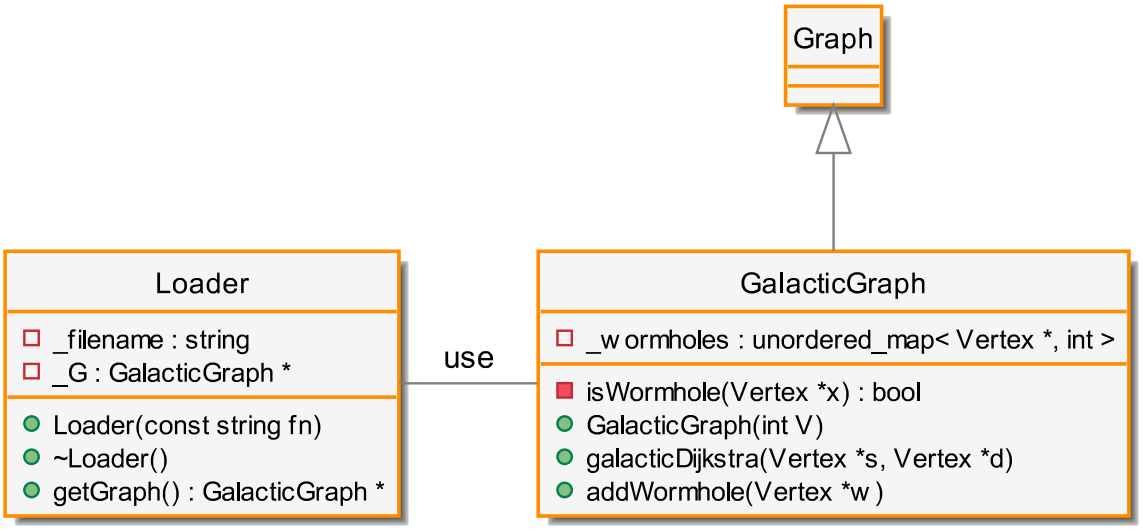

Figura 2.3: Loader e Grago galattico

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 $O(2(V + E)2log_2V)$ che diventa $O(2E2log_2V)$ se ogni vertice è raggiungibile dalla sorgente, ma poichè è possibile portare le costanti moltiplicative fuori diremo che la complessità finale sarà $\mathbf{O(Elog_2V)}$. 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 $O(log_2V)$. Ciò avrebbe comportato un miglioramento nel caso in cui ci fossero stati molti archi, avendo un costo asintotico pari a $O(Vlog_2V + E)$.

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.

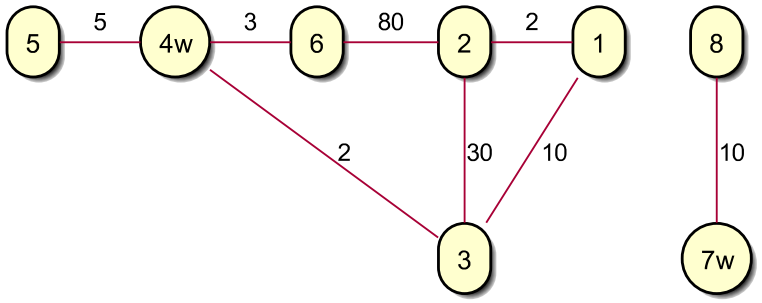

Figura 2.4: Grafo non connesso.

```
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
```

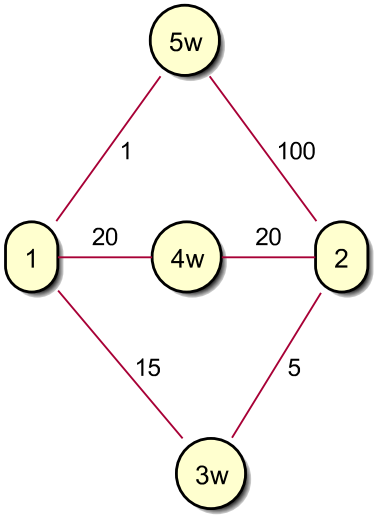

Figura 2.5: Grafo a diamante.

```
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 destizione è lo stesso, infatti il nodo 11 è presente in entrambi i cammini minimi.

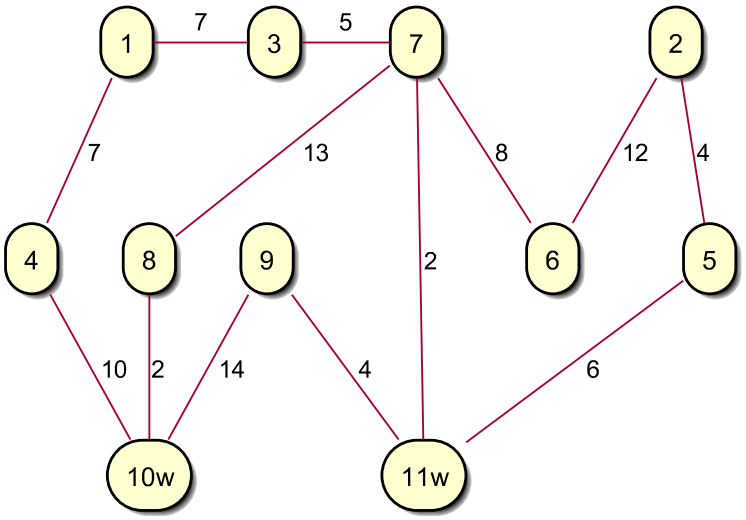

Figura 2.6: Grafo d'esempio.

```
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 contentente 317080 nodi, 1049866 archi e 10 wormholes, ovviamente previo commento nell'implementazione del parser circa i controlli sull'input. Successimante 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

```
#ifndef _HEAP_HPP_
#define _HEAP_HPP_

#include <iostream>
#include <algorithm>
#include <utility>
#include <deque>

/** Heap Data Structure */
template <class T> // object inside this DS are templates
class Heap {
  private:
    int heapSize;

  protected:// made attributes protected 'cause
            // we need in Priority Queues
    std::deque< std::pair<T, int> > A;

  public:
    Heap() : heapSize(0){};
    ~Heap(){};

    int find(T el);

    // inserting elements using std::pair
    void insert(T el, int key); // must use Int for key and T for obj to
     ↪  store.


    int parent(int i){return (i-1)/2;};
    int left(int i){return (2*i + 1);};
    int right(int i){return (2*i + 2);};

    /**
     * override of deque positional operators
     * must use this for access at deque (e.g. Priority Queue)
     * return std::pair from position at i
    */
    std::pair<T, int> at(int i);

    /**
     * override of deque positional operators
     * must use this for access at deque (e.g. Priority Queue)
     * set pair<key, T element> at position i
     */
    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_
```

```cpp
#include <priorityqueue/heap/heap.hpp>

template <class T>
void Heap<T>::insert(T el, int key) {
  this->A.push_back( std::make_pair(el, key) );
}

template <class T>
std::pair<T, int> Heap<T>::at(int i) {
  return this->A.at(i);
}

template <class T>
void Heap<T>::set(int i, std::pair<T, int> el) {
  this->A.at(i) = el;
}

template <class T>
void Heap<T>::swap(int i, int j){
  std::iter_swap(this->A.begin()+i, this->A.begin()+j);
}

template <class T>
void Heap<T>::popBack(){
  this->A.pop_back();
}

template <class T>
void Heap<T>::resize(int x){
  this->A.resize(x);
}


template <class T>
void Heap<T>::postOrder(int index){
  if(index < Heap<T>::getHeapSize()){
    postOrder(Heap<T>::left(index));
    postOrder(Heap<T>::right(index));
    std::cout << this->A.at(index).first << " ";
  }
}


template <class T>
void Heap<T>::printHeap(){
  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  }
```

```
#ifndef _MINHEAP_HPP_
#define _MINHEAP_HPP_

#include <priorityqueue/heap/heap.hpp>

/** MinHeap using all attributes and methods from Heap */
template <class T>
class MinHeap : public Heap<T> {
  public:
    /** build a MinHeap from elements inserted in A[n] in O(n) */
    void buildHeap();

    /** Min heapify procedure makes magic, threating A[n] as a BT.
     * O(log n) */
    void heapify(int i);

    /** creasily (Heap)sorting A[n] in O(n*logn) */
    void heapSort();
};

#endif //_MINHEAP_HPP_
```

```
#include <priorityqueue/heap/minHeap.hpp>
#include "heap.cpp"

template <class T>
void MinHeap<T>::heapify(int i){
  if( i >= 0 ){
    int min = 0;
    int l = Heap<T>::left(i);
    int r = Heap<T>::right(i);

    //check if l exist and then look for a min
    if( l <= this->getHeapSize()-1 && \
        this->A.at(i).second > this->A.at(l).second )
      min = l;
    else
      min = i;

    // if not l neither i is min t then r is
    if( r <= this->getHeapSize()-1 && \
        this->A.at(min).second > this->A.at(r).second )
      min = r;

    //if max is not i then swap and re-run heapify recusively
    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  }
```

```cpp
#ifndef _MINPRIORITYQUEUE_HPP_
#define _MINPRIORITYQUEUE_HPP_

#include <priorityqueue/heap/minHeap.hpp>

/**
 * min priority queue created using a Min Heap
 */
template <class T>
class MinPriorityQueue {
  private:
    MinHeap<T> A; // used for operations

  public:
    MinPriorityQueue(){};
    ~MinPriorityQueue(){};

    /** Insert an element with his key in O(log n) */
    void insert(T el, int key);

    /** get minimum in O(1) */
    std::pair<T, int> minimum();

    /** Extract the minimum in O(log n) */
    std::pair<T, int> extractMin();

    /** decrease the priority of an element by a key value */
    void decreasePriority(T el, int key);

    /** Printing the priority queue */
    void print(){this->A.printHeap();};
    void printAll(){this->A.printA();};

    /** return size of priority queue */
    int getPQSize(){return A.getHeapSize();};
};

#endif // _MINPRIORITYQUEUE_HPP_
```

```cpp
#include <priorityqueue/minPriorityQueue.hpp>
#include <other/debug.hpp>

#include "heap/minHeap.cpp"

#include <cassert>

```

```
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 >
       ↪  this->A.at(i).second ) {
62      this->A.swap(this->A.parent(i), i);
63      i = this->A.parent(i);
64    }
65  }
```

```
#ifndef _GALACTICTGRAPH_HPP_
#define _GALACTICTGRAPH_HPP_

#include <unordered_map>
#include <graph/graph.hpp>

/**
 * @brief GalacticGraph. Using wormholes for fast travel.
 * if there is a wormhole near s and d,
 * use that to teleport in 1 unit time.
 */
class GalacticGraph : public Graph {
  private:
    std::unordered_map<Vertex *, int> _wormholes;
    bool isWormhole(Vertex *x); // return true if a node is in hashmap

  public:
    GalacticGraph(int V) : Graph(V) {};

    // compute path from s to d, then from s to first worm and reapply
     ↪  dijkstra
    // using d as source, and get path from d to worm
    void galacticDijkstra(Vertex *s, Vertex *d);

    // load a vertex in hashmap
    void addWormhole(Vertex *w){this->_wormholes.insert(std::make_pair(w,
     ↪  1));};
};

#endif //_GALACTICTGRAPH_HPP_
```

```
#include <galacticgraph.hpp>


bool GalacticGraph::isWormhole(Vertex *x) {
  if(x != nullptr)
    return this->_wormholes.find(x) != this->_wormholes.end();
  // O(1) search
  return false;
}


void GalacticGraph::galacticDijkstra(Vertex *s, Vertex *d) {
  auto wFound = new Vertices();

  // 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 womrhole in path, get the first one extracted
44      auto w2 = wFound->empty() ? nullptr : wFound->front();
45
46      if(w2 != nullptr && w1 != w2) { // if womrhole 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  }
```

```hpp
#ifndef _LOADER_HPP_
#define _LOADER_HPP_

#include <string>
#include <galacticgraph.hpp>

/**
 * @brief Loader class for parse file.txt into Graph
 */
class Loader {
  private:
    std::string _filename;
    GalacticGraph *_G = nullptr;

  public:
    Loader(const std::string fn);
    ~Loader(){ delete _G;}

    GalacticGraph *getGraph(){return this->_G;}
};

#endif //_LOADER_HPP_
```

```cpp
#include <iostream>
#include <fstream>

#include <loader.hpp>

Loader::Loader(const std::string fn) : _filename(fn) {

  std::ifstream filestream(this->_filename);
  if(filestream.fail()){
    std::cerr << "Error opening file" << std::endl;
    exit(EXIT_FAILURE);
  }

  int V, E, W;
  filestream >> V;
  filestream >> E;
  filestream >> W;

  if(V < 1 || V > 1000 ) {
    std::cerr << "Solar systems must be at least 2 and less then 1k " <<
    ↪  std::endl;
    exit(EXIT_FAILURE);
  }
```

```
23
24    if(E < 0 || E > 10000 ) {
25      std::cerr << "Connections must be at least one and less then 10k" <<
          ↪  std::endl;
26      exit(EXIT_FAILURE);
27    }
28
29    if(W < 1 || W > V) {
30      std::cerr << "Wormholes must be at least 2 and less then V("<<V<<")"
          ↪  << std::endl;
31      exit(EXIT_FAILURE);
32    }
33
34    std::cout << "Filling a graph with "<< V<<" nodes, ";
35    std::cout << E <<" edges, ";
36    std::cout << W <<" wormholes\n";
37
38    this->_G = new GalacticGraph(V);
39    auto vertices = this->_G->getVertices();
40
41    IFDEBUG debug_print << "STARTING READING FILE: " <<E-W<< std::endl;
42    for(auto it = 0; it < E-W; it++) {
43      int a, b, w;
44      filestream >> a;
45      filestream >> b;
46      filestream >> w;
47      this->_G->addEdge(vertices->at(a-1), vertices->at(b-1), w);
48    }
49
50
51    IFDEBUG debug_print << "STARTING READING 2ndhalf: "<<W<< std::endl;
52    for(auto it = 0; it < W; it++) {
53       int a, b, w;
54       filestream >> a;
55       filestream >> b;
56       filestream >> w;
57       this->_G->addEdge(vertices->at(a-1), vertices->at(b-1), w);
58       this->_G->addWormhole(vertices->at(b-1));
59    }
60
61    filestream.close();
62    return;
63  }
```

```
#ifndef _DEBUG_HPP_
#define _DEBUG_HPP_

  // debugging
  #include <iostream>
  #define debug_print std::cerr

  #ifdef DEBUG
    #define IFDEBUG if(1)
  #else
    #define IFDEBUG if(0)
  #endif

#endif // _DEBUG_HPP
```

```
#ifndef _ITEM_HPP_
#define _ITEM_HPP_

#include <limits>

/**
 * @brief Generic Item
 * @details using a double as a key
 * @tparam Data parameter as value
 */
template <class Data>
class Item {
  private:
    double _key;
    Data _data;

  public:
    //Item() : _key(std::numeric_limits<int>::min()), _data {} {}
    Item(double k) : _key(k), _data {} {}
    Item(double k, Data &d) : _key(k), _data(d) {}

    //generic setter and getter
    void setKey(double k) { this->_key = k; }
    double getKey() { return this->_key; }

    void setData(Data &data) { this->_data = data; }
    Data& getData() { return this->_data; }
};

#endif //_ITEM_HPP_
```

```
#ifndef _VERTEX_HPP_
#define _VERTEX_HPP_

#include <limits>
#include <utility> //make_pair()
#include <unordered_map>

#include <other/debug.hpp>
#include <other/item.hpp>

// dijkstra algorithm
#define inf std::numeric_limits<int>::max()

// weighted node
#define WeightedVertex std::pair< Vertex *, int >

// using adjacent map instead of vector -> O(1) direct access
#define AdjacentMap std::unordered_map< Vertex *, int >


/**
 * @brief Vertex class inherites from generic Item
 * with double as key and void* as data.
 */
class Vertex : public Item<void *>{
  private:
    double _d = inf; // distance
    Vertex *_prev = nullptr; // parent
    AdjacentMap _adjmap; // map

  public:
    Vertex(int key) : Item<void *>(key) {}
    ~Vertex(){}

    void setParent(Vertex *p) { this->_prev = p; }
    Vertex* getParent() { return this->_prev;}

    void setD(double d) { this->_d = d; }
    double getD() { return this->_d; }

    // add edge to a node in map
    void addEdge(WeightedVertex u);

    void printAdj();
    AdjacentMap getAdj() { return this->_adjmap; };

    // overload of operator for comparing situation
```

```
48      bool operator== (Vertex &other) {return this->getKey() ==
        ↪  other.getKey();}
49      bool operator!= (Vertex &other) {return this->getKey() !=
        ↪  other.getKey();}
50
51      // printing purpouse
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_
```

```
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  }
```

```cpp
#ifndef _GRAPH_HPP_
#define _GRAPH_HPP_

#include <list>
#include <vector>
#include <functional>

#include <graph/vertex.hpp>
#include <priorityqueue/minPriorityQueue.hpp>

#define Vertices std::vector<Vertex *>

/**
 * @brief Undirected Graph class
 *
 */
class Graph {
  private:
    const int _V; // no need to modify once set
    Vertices *_vertices;

  public:
    Graph(int vertices);
    ~Graph(){ delete [] _vertices; }

    // return all vertex
    Vertices* getVertices() {return _vertices;}

    //get single vertex if correctly indexed
    Vertex* getVertex(int i) { return (i < (int)this->_vertices->size() ? this->_vertices->at(i) : nullptr) ;}

    void printGraph();

    // add two Edge per call (undirected)
    void addEdge(Vertex *a, Vertex *b, int weight);

    /* Apply dijkstra algorithm starting from a node */
    void dijkstra(Vertex *s);

    /* overloaded dijkstra -> get a lambda function for stop criteria after
     * extraced node finished relaxing. Return extracted node. */
    Vertex* dijkstra(Vertex *s, std::function <bool (Vertex *)>const& lambda);

    /* return a path from node D up to last prev (nullptr) or S */
    std::vector<Vertex> getPath(Vertex *s, Vertex *d);
```

```
46
47      /* print path */
48      int printPath(Vertex *s, Vertex *d, bool print = false, bool reverse =
        ↪  false);
49      int printPath(Vertex *s, Vertex *d, std::vector<Vertex> &path, bool
        ↪  print = false, bool reverse = false);
50  };
51
52  #endif //_GRAPH_HPP_
```

```
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      Q->insert((*v), (*v)->getD()); // insert all vertices
44
45    while( Q->getPQSize() != 0 ) {
46      Vertex *u = Q->extractMin().first;
47      auto uAdj = u->getAdj();
48      for(auto v : uAdj)
49        relax(u, v.first, v.second, Q);
50    }
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 }
```

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 LaTeX, *Lorenzo Pantieri*
http://www.lorenzopantieri.net/LaTeX.html

[5] LaTeX/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/