```hpp
#ifndef DICTIONNAIRE_HPP
#define DICTIONNAIRE_HPP

class dictionnaire {

public:
  //
  virtual double rechercher(const int l, const int c) const =0;
  virtual void ajouter(const int l, const int c, const double v) =0;
  virtual void supprimer(const int l, const int c) =0;
  virtual int longueur() const =0;
};

#endif
```



```hpp
#ifndef DICTIONNAIREL_HPP
#define DICTIONNAIREL_HPP

#include "triplet.hpp"
#include "noeud.hpp"
#include "dictionnaire.hpp"

class dictionnaireL : public dictionnaire {

protected:
  noeud<triplet> *liste;
  int lg; // longueur du dictionnaireL

  void cloner(const dictionnaireL &d) {
    noeud<triplet> *n = d.liste;
    this->liste = nullptr;
    if (n!=nullptr) {
      this->liste = new noeud<triplet>(n->getElement());
      noeud<triplet> *q = this->liste;
      while (n->getSuivant() != nullptr) {
        n = n->getSuivant();
        q->setSuivant(new noeud<triplet>(n->getElement()));
        q = q->getSuivant();
      }
    }
  }

  /*
   * Rôle : désalloue le noeud n et ses suivants
   */
  void detruire(noeud<triplet> *n) {
    if (n!=nullptr) {
      detruire(n->getSuivant());
      delete n;
    }
  }

  /*
   * Rôle : renvoie le noeud qui contient le triplet de valeur
   *        ligne/colonne = (l,c)
   */
  noeud<triplet> *getTriplet(const int l, const int c) const {
    noeud<triplet> *p = this->liste;
    while (p != nullptr) {
      triplet t = p->getElement();
      if (t.getLigne() == l && t.getColonne() == c)
        // trouvé !
        return p;
      // else passer au triplet suivant
      p = p->getSuivant();
    }
    return nullptr;
  }

public:
  // constructeur
  dictionnaireL() : liste(nullptr), lg(0) {}
  // constructeur de copie
  dictionnaireL(const dictionnaireL &d) {
    this->cloner(d);
  }
  // destructeur
  ~dictionnaireL() { this->detruire(this->liste); }

  // surcharge affectation
  dictionnaireL &operator=(const dictionnaireL &d) {
```

```
    this->cloner(d);
    return *this;
  }


  /*
   * Rôle : supprime le triplet (l,c,0) dans le dictionnaire
   *        s'il est présent
   */
  void supprimer(const int l, const int c) override {
    noeud<triplet> *p = this->liste, *q = nullptr;
    bool trouve = false;
    while (p != nullptr && !trouve) {
      triplet t = p->getElement();
      if (t.getLigne() == l && t.getColonne() == c)
        trouve = true;
      else { // passer au triplet suivant
        q = p;
        p = p->getSuivant();
      }
    }
    if (trouve) { // supprimer l'élément p
      if (q==nullptr)
        // le 1er de la liste
        this->liste->setSuivant(p->getSuivant());
      else
        q->setSuivant(p->getSuivant());
      delete p;
    }
  }

  /*
   * Rôle : recherche dans le dictionnaire le triplet
   *        de ligne l et colonne c et renvoie sa valeur réelle
   *        si trouvé, sinon renvoie 0.0
   */
  double rechercher(const int l, const int c) const override {
    noeud<triplet> *p = this->getTriplet(l,c);
    return p==nullptr ? 0.0 : p->getElement().getValeur();
  }

  /*
   * Rôle : ajoute le triplet (l,c,v) dans le dictionnaire
   *        s'il n'est pas déjà présent"
   */
  void ajouter(const int l, const int c, const double v) override {
    noeud<triplet> *p = this->getTriplet(l,c);
    if (p!=nullptr) {
      // modifier l'élément
      triplet &tp = p->getElement();
      tp.setValeur(v);
    }
    else {
      // p == nullptr
      // l'ajouter en tête dans la liste de triplets
      p = new noeud<triplet>(triplet(l,c,v));
      p->setSuivant(this->liste);
      this->liste = p;
      lg++;
    }
  }

  /*
   * Rôle : renvoie la longueur du dictionnaire courant
   */
  int longueur() const override { return lg; }
```



```
};

#endif
```

```hpp
#ifndef DICTIONNAIREM_HPP
#define DICTIONNAIREM_HPP

#include <map>
#include "dictionnaire.hpp"

/*
 * La classe "indices" représente les indices (lignes, colonnes) d'une
 * valeur de la matrice creuse.
 *
 * @author Vincent Granet (vg@unice.fr)
 */
class indices {
private:
  int l, c; //
public:
  // constructeurs
  indices(const int i, const int j) : l(i), c(j) {}
  // accesseurs
  int getLigne()   const { return this->l; }
  int getColonne() const { return this->c; }
  // mutateurs
  void setLigne(const double l)   { this->l = l ; }
  void setColonne(const double c) { this->c = c; }

  // opérateurs de comparaisons
  bool operator==(const indices & c) const {
    return  this->l == c.l  && this->c == c.c;
  }

  bool operator!=(const indices & c) const {
    return !(*this == c);
  }

  bool operator<(const indices & c) const {
    if (this->l < c.l) return true;
    if (this->l > c.l) return false;
    // this->l == c.l
    return  this->c < c.c;
  }

  bool operator>(const indices & c) const {
    if (this->l > c.l) return true;
    if (this->l < c.l) return false;
    // this->l == c.l
    return  this->c > c.c;
  }

  bool operator<=(const indices & c) const {
    return !(*this>c);
  }

  bool operator>=(const indices & c) const {
    return !(*this<c);
  }
};

class dictionnaireM : public dictionnaire {

protected:
   std::map<indices, double> lesElements;

public:
   /*
   * Rôle : recherche dans le dictionnaire le triplet
   *        de ligne l et colonne c et renvoie sa valeur réelle
```



```hpp
   *        si trouvé, sinon renvoie 0.0
   */
  double rechercher(const int l, const int c) const override {
    //
    auto v = lesElements.find(indices(l,c));
    return v != lesElements.end() ? v->second : 0.0;
  }

  /*
   * Rôle : supprime le triplet (l,c,0) dans le dictionnaire
   *        s'il est présent
   */
  void supprimer(const int l, const int c) override {
    indices ij(l,c);
    auto x = lesElements.find(ij);
    if (x != lesElements.end()) {
      // il existe déjà une valeur d'indices ij
      // la supprimer
      lesElements.erase(x);
    }
  }

  /*
   * Rôle : ajoute le triplet (l,c,v) dans le dictionnaire
   *        s'il n'est pas déjà présent"
   */
  void ajouter(const int l, const int c, const double v) override {
    // rechercher le triplet correspondant dans le vecteur
    indices ij(l,c);
    auto x = lesElements.find(ij);
    if (x != lesElements.end())
      // il existe déjà une  valeur leur indices ij
      x->second = v;
    else
        lesElements[ij] = v;
  }

  /*
   * Rôle : renvoie la longueur du dictionnaire courant
   */
  int longueur() const override { return lesElements.size(); }
};

#endif
```

```
#ifndef DICTIONNAIREV_HPP
#define DICTIONNAIREV_HPP

#include <vector>
#include "triplet.hpp"
#include "dictionnaire.hpp"

class dictionnaireV  : public dictionnaire {

protected:
  std::vector<triplet> lesElements;

public:
  /*
   * Rôle : recherche dans le dictionnaire le triplet
   *        de ligne l et colonne c et renvoie sa valeur réelle
   *        si trouvé, sinon renvoie 0.0
   */
  double rechercher(const int l, const int c) const override {
    //
    for (triplet t : this->lesElements)
      if (t.getLigne()==l && t.getColonne()==c)
        // trouvé !
        return t.getValeur();
    // triplet non trouvé
    return 0.0;
  }

  /*
   * Rôle : supprime le triplet (l,c,0) dans le dictionnaire
   *        s'il est présent
   */
  void supprimer(const int l, const int c) override {
    bool trouve = false;
    int i=0;
    while (i<lesElements.size() && !trouve) {
      triplet t = lesElements[i];
      if (t.getLigne()==l && t.getColonne()==c)
        trouve=true;
      else i++;
    }
    //
    if (trouve)
      lesElements.erase(lesElements.begin()+i, lesElements.begin()+i+1);
  }

  /*
   * Rôle : ajoute le triplet (l,c,v) dans le dictionnaire
   *        s'il n'est pas déjà présent"
   */
  void ajouter(const int l, const int c, const double v) override {
    // rechercher le triplet correspondant dans le vecteur
    bool trouve = false;
    int i=0;
    while (i<lesElements.size() && !trouve) {
      triplet t = lesElements[i];
      if (t.getLigne()==l && t.getColonne()==c)
        trouve=true;
      else i++;
    }
    //
    if (trouve)
      lesElements[i].setValeur(v);
    else
      // non trouvé => l'ajouter dans la liste de triplets
      lesElements.push_back(triplet(l,c,v));
```



```
  }

  /*
   * Rôle : renvoie la longueur du dictionnaire courant
   */
  int longueur() const override { return lesElements.size(); }
};

#endif
```

```hpp
#ifndef INDEXEXCEPTION_HPP
#define INDEXEXCEPTION_HPP

#include <exception>
#include <cstring>
#include <string>
/*
 * Hiérarchie d'exception pour gérer les erreurs d'indexation à un
 * élément d'un tableau représenté par la classe générique « tableau »
 */
class IndexException : public std::exception {
protected:
  std::string msg;
public:
  IndexException(int i, int j) {
    msg = "(" + std::to_string(i) + "," +std::to_string(j) +
      ") : Index out of bounds exception";

  }
  // redéfinition de what
  const char* what() const noexcept override {
    return msg.c_str();
  }
};

#endif
```



```
DICO=LISTE                # par défaut
CC = g++                  # le compilateur à utiliser
CFLAGS = -D$(DICO) --std=c++14    # les options du compilateu
LDFLAGS =                 # les options pour l'éditeur de liens
SRC = test.cpp
PROG = test               # nom de l'exécutable
OBJS =  $(SRC:.cpp=.o) # les .o qui en découlent
.SUFFIXES: .cpp .o        # lien entre les suffixes

all: $(PROG)

# étapes de compilation et d'édition de liens
# $@ la cible  $^ toutes les dépendances
$(PROG): $(OBJS)
        $(CC) -o $@ $^ $(LDFLAGS)

test.o: MatriceAbstraite.hpp Matrice.hpp MatriceCreuse.hpp \
        triplet.hpp noeud.hpp dictionnaire.hpp \
        dictionnaireM.hpp \
        dictionnaireV.hpp \
        dictionnaireL.hpp

# le lien entre .o et .c
# $< dernière dépendance
%.o: %.cpp
        $(CC) $(CFLAGS) -c $<

# pour faire propre
.PHONY: clean
clean:
        rm -f *.o *~ core *.gch a.out $(PROG)
```

```hpp
#ifndef MATRICEABSTRAITE_HPP
#define MATRICEABSTRAITE_HPP

#include <cassert>
#include <iostream>

/*
 * La classe MatriceAbstraite
 *
 * @author Vincent Granet (vg@unice.fr)
 */
template <int M, int N> class MatriceAbstraite {

protected:
  /*
   * Rôle : *this = m
   */
  void copie(const MatriceAbstraite<M,N> &m) {
    for (int i=0; i<M; i++)
      for (int j=0; j<N; j++)
        this->setValeur(i, j, m.getValeur(i,j));
  }

public:
  MatriceAbstraite() { assert(M>0 && N>0); }

  // antécédent :  l>=0 et l<nbLignes et  c>=0 et c<nbColonnes
  // rôle : renvoie le réel de coordonnées (l,c) dans la matrice courante
  virtual double getValeur(const int l, const int c) const=0;

  // antécédent :  l>=0 et l<nbLignes et  c>=0 et c<nbColonnes
  // rôle : affecte le réel v de coordonnées (l,c) dans la matrice courante
  virtual void  setValeur(const int l, const int c, const double v)=0;

  // surcharge de l'opérateur de sortie << pour écrire la matrice abstraite
  // m sur l'ostream f
  // surcharge de l'opérateur de sortie << pour écrire la matrice
  // abstraite p sur l'ostream f
  friend std::ostream& operator<<(std::ostream &f, const MatriceAbstraite<M,N> &
m) {
    for (int i=0; i<M; i++) {
      for (int j=0; j<N; j++)
        f << m.getValeur(i,j) << " ";
      //
      f << std::endl;
    }
    return f;
  }

  /*
   * Rôle : initialise la matrice carrée courante abstraite à la matrice
   *        identité
   */
  void identite() {
    assert(M==N);
    // mettre les 0
    for (int i=0; i<M; i++) {
      for (int j=0; j<N; j++) this->setValeur(i, j, 0.0);
      // mettre les 1
      this->setValeur(i, i, 1.0);
    }
  }
};

#endif
```



```hpp
#ifndef MATRICECREUSE_HPP
#define MATRICECREUSE_HPP

#include "MatriceAbstraite.hpp"
#ifdef LISTE
#include "dictionnaireL.hpp"
#elif VECTEUR
#include "dictionnaireV.hpp"
#else // MAP par défaut
#include "dictionnaireM.hpp"
#endif

#include "IndexException.hpp"

/*
 * La classe MatriceCreuse représente des matrices creuses, i.e. des
 * matrices dont la majorité des valeurs sont nulles. Seules les
 * valeurs non nulles son mémorisées.
 *
 * @author Vincent Granet (vg@unice.fr)
 */
template <int M, int N>
class MatriceCreuse : public MatriceAbstraite<M,N> {
private:
  typedef MatriceAbstraite<M,N> super;

protected:
  dictionnaire *lesElements;

public:
  // constructeur
  MatriceCreuse() : MatriceAbstraite<M,N>() {
#ifdef LISTE
    lesElements = new dictionnaireL();
#elif VECTEUR
    lesElements = new dictionnaireV();
#else // MAP
    lesElements = new dictionnaireM();
#endif
  }

  // constructeur de copie
  MatriceCreuse(const MatriceAbstraite<M,N> &m) { super::copie(m); }

  // surcharge affectation
  MatriceCreuse<M,N> &operator=(const MatriceAbstraite<M,N> &m) {
    super::copie(m);
    return *this;
  }

  // Antécédent : l>=0 et l<nbLignes et  c>=0 et c<nbColonnes
  // Rôle : renvoie le réel de coordonnées (l,c) dans la matrice courante
  double getValeur(const int l, const int c) const override {
    if (l<0 || l>=M || c<0 || c>=N)
      throw IndexException(l,c);
    return this->lesElements->rechercher(l,c);
  }

  // Antécédent :  l>=0 et l<nbLignes et  c>=0 et c<nbColonnes
  // Rôle : affecte le réel v de coordonnées (l,c) dans la matrice courante
  void setValeur(const int l, const int c, const double v) override {
    if (l<0 || l>=M || c<0 || c>=N)
      throw IndexException(l,c);
    // ajouter la valeur v dans la matrice creuse si elle est différente de 0
    if (v!=0)
      this->lesElements->ajouter(l,c,v);
```

```
    else
      // v=0 et s'il existe, on supprime le triplet dans le dictionnaire
      this->lesElements->supprimer(l,c);
  }

  // Rôle : renvoie le produit *this * m
  template <int P>
  MatriceAbstraite<M,P> *produit(const MatriceAbstraite<N,P> &m) const {
    MatriceAbstraite<M,P> *r = new MatriceCreuse<M,P>();
    for (int i = 0; i < M; i++)
      for (int j = 0; j < P; j++) {
        double  somme = 0.0;
        for (int k = 0; k < N; k++)
          somme += this->getValeur(i, k) * m.getValeur(k, j);
        // r(i,j) = somme de k = 0 à N des this(i,k)*m(k,j)
        r->setValeur(i, j, somme);
      }
    return r;
  }

  // Rôle : surcharge de l'opérateur * pour faire le produit *this * m
  template <int P>
  MatriceAbstraite<M,P> *operator*(const MatriceAbstraite<N,P> &m) const {
    return this->produit(m);
  }

  //
  // Rôle : renvoie le taux de remplissage t de la matrice 0<=t<=1
  double tauxRemplissage() {
    return  this->lesElements->longueur()/(double) (M*N);
  }

};

#endif
```



```
#ifndef MATRICE_HPP
#define MATRICE_HPP

#include "IndexException.hpp"
#include "MatriceAbstraite.hpp"

/*
 * La classe Matrice représente des matrices
 *
 * @author Vincent Granet (vg@unice.fr)
 */
template <int M, int N>
class Matrice : public MatriceAbstraite<M,N> {

private:
  typedef MatriceAbstraite<M,N> super;

protected:
  double lesElements[M][N] = {};

public:
  // constructeur
  Matrice() : MatriceAbstraite<M,N>() {}

  // constructeur de copie
  Matrice(const MatriceAbstraite<M,N> &m) { super::copie(m); }

  // surcharge affectation
  Matrice<M,N> &operator=(const MatriceAbstraite<M,N> &m) {
    super::copie(m);
    return *this;
  }
  // Antécédent :  l>=0 et l<nbLignes et  c>=0 et c<nbColonnes
  // Rôle : renvoie le réel de coordonnées (l,c) dans la matrice courante
  double getValeur(const int l, const int c) const override {
    if (l<0 || l>=M || c<0 || c>=N)
      throw IndexException(l,c);
    return this->lesElements[l][c];
  }

  // Antécédent :  l>=0 et l<nbLignes et  c>=0 et c<nbColonnes
  // Rôle : affecte le réel v de coordonnées (l,c) dans la matrice courante
  void setValeur(const int l, const int c, const double v) override {
    if (l<0 || l>=M || c<0 || c>=N)
      throw IndexException(l,c);
    this->lesElements[l][c] = v;
  }

  // Rôle : renvoie le produit *this * m
  template <int P>
  MatriceAbstraite<M,P> *produit(const MatriceAbstraite<N,P> &m) const {
    MatriceAbstraite<M,P> *r = new Matrice<M,P>();
    for (int i = 0; i < M; i++)
      for (int j = 0; j < P; j++) {
        double  somme = 0.0;
        for (int k = 0; k < N; k++)
          somme += this->getValeur(i, k) * m.getValeur(k, j);
        // r(i,j) = somme de k = 0 à N des this(i,k)*m(k,j)
        r->setValeur(i, j, somme);
      }
    return r;
  }

 // Rôle : surcharge de l'opérateur * pour faire le produit *this * m
  template <int P>
  MatriceAbstraite<M,P> *operator*(const MatriceAbstraite<N,P> &m) const {
```

```
    return this->produit(m);
  }
};

#endif
```



```
#ifndef NOEUD_HPP
#define NOEUD_HPP

/*
 * la classe générique noeud représente un maillon d'une liste chaînée
 *
 * @author Vincent Granet (vg@unice.fr)
 */

template <typename T>
class noeud {

private:
  T elt;
  noeud<T> *suivant;

public:
  // constructeur
  noeud(T e, noeud<T> *s = nullptr) : elt(e), suivant(s) {}

  // accesseurs
  const T & getElement() const {
    return this->elt;
  }

  T & getElement() {
    return this->elt;
  }

  noeud<T> *getSuivant() const {
    return this->suivant;
  }
  // mutateurs
  void setSuivant(noeud<T> *s) {
    this->suivant = s;
  }

  void setElement(const T &x) {
    this->elt = x;
  }
};

#endif
```

```cpp
#ifndef TRIPLET_HPP
#define TRIPLET_HPP

#include <cassert>

/*
 * La classe triplet représente la valeur d'une matrice creuse de
 * réels.  Elle contient la valeur du réel et ses coordonnées (lignes,
 * colonnes) dans la matrice creuse.
 *
 * @author Vincent Granet (vg@unice.fr)
 */
class triplet {
private:
  int l, c; //
  double valeur;
public:
  // constructeurs
  triplet(const int i, const int j, const double v) :
    l(i), c(j) , valeur(v)  { assert(v!=0); }
  // accesseurs
  double getValeur() const { return this->valeur; }
  int getLigne()     const { return this->l; }
  int getColonne()   const { return this->c; }
  // mutateurs
  void setValeur(const double v)  {
    assert(v!=0);
    this->valeur = v;
  }
  void setLigne(const int l)   { this->l = l ; }
  void setColonne(const int c) { this->c = c; }
};

#endif
```