Mise en oeuvre de la classe générique Graphe<S,T>

1. Organisation générale

L'objectif est de réaliser la mise en oeuvre d'un graphe de la manière la plus générale possible.

Un graphe est classiquement défini par un ensemble de sommets et par un ensemble d'arêtes où une arête représente une relation entre deux sommets.

Au sens le plus général, seule cette information topologique "est relié à" est prise en compte dans un graphe.
A chaque sommet est généralement associée une information (au moins un identifiant), de même qu'à chaque arête est généralement associée une information (identifiant, poids, etc.).

Pour rester le plus général possible, nous faisons donc l'hypothèse qu'une information de type *T*, respectivement de type *S* est inscrite dans chaque sommet, respectivement dans chaque arête.

Nous aboutissons donc aux définitions de classes suivantes :

- ***Sommet<T>*** : classe template représentant un sommet, munie de trois attributs : *clef* (identifiant), *degré* et *v* de type *T* qui est l'information notée au niveau du sommet.

- ***Arete<S,T>*** : classe template représentant une arête, munie de quatre attributs : *clef* (identifiant), *début* qui est un pointeur sur le *Sommet<T>* qui constitue l'extrémité initiale de l'arête, *fin* qui est un pointeur sur le *Sommet<T>* qui constitue l'extrémité finale de l'arête et enfin *v* de type *S* qui est l'information notée au niveau de l'arête. Il y a là une forte analogie avec les bases de données relationnelles : *début* et *fin* peuvent être interprétées comme des clefs étrangère vers une table de sommets.

Afin de factoriser le code commun à ces deux classes, on définit encore :

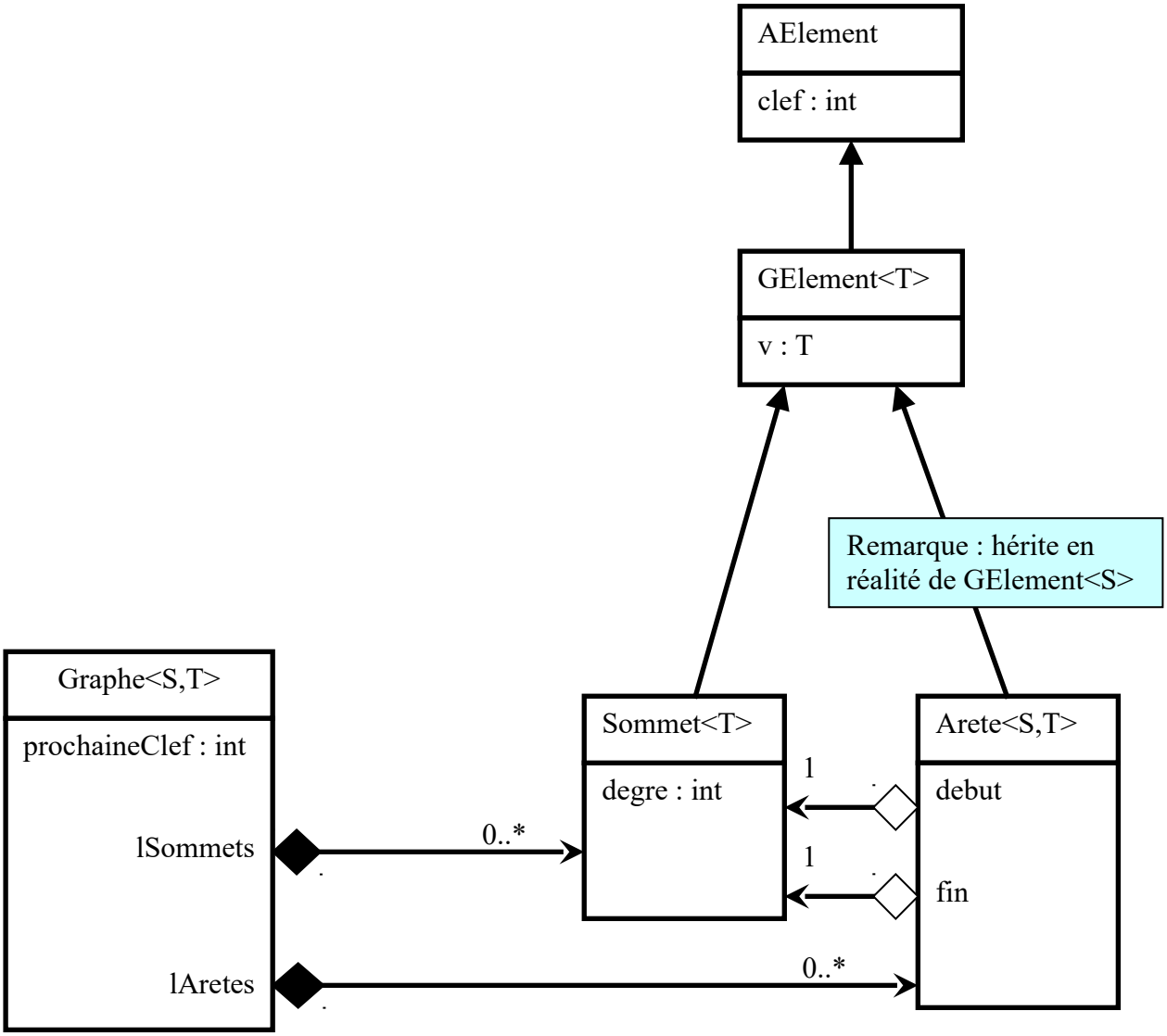
- ***AElement*** : la classe (non template) qui permet de mutualiser l'attribut *clef*

- ***GElement<T>*** : la classe template qui permet de factoriser l'attribut *v*.

Nous définissons enfin la classe template (générique) :

- ***Graphe<S,T>*** représentant un graphe quelconque où *T* est la nature des informations associées aux sommets et *S* celle des informations associées aux arêtes. Un graphe est défini par une liste de sommets, par une liste d'arêtes et par un générateur de clefs primaires.

L'organisation des classes *AElement*, *GElement<T>*, *Sommet<T>*, *Arete<S,T>* et *Graphe<S,T>* peut être résumée par le diagramme suivant :



Un exemple de graphe et de la structure correspondante créée en mémoire figure à la fin de ce document (cf. paragraphe 6.15).

2. Mise en oeuvre de la classe (non générique) AElement

AElement (pour *Ancêtre-Elément*) est la classe de base de *GElement<T>*.
Elle permet de factoriser la notion de clef. *AElement* est donc une classe dont le seul attribut est un entier *clef* servant de clef primaire.
Il faut donc :
Ecrire la classe *AElement* ainsi définie. Pour simplifier, le membre *clef* peut être public.
Munir *AElement* d'un constructeur, d'un opérateur de conversion en *string* (fonction membre dont la signature est : ***operator*** *string () const*) et de l'opérateur << d'écriture sur un flux (cette dernière méthode est une fonction ordinaire, elle peut être *friend*, elle peut avantageusement se servir de l'opérateur de conversion en *string*).

Il est inutile d'écrire destructeur, constructeur de copie ou getters et setters.
Ecrire (dans un fichier *TestGrapheGeneral.cpp*) une fonction *main*() de test des fonctionnalités des classes *AElement*, *GElement<T>, Sommet<T>*, *Arete<S,T>* et *Graphe<S,T>*. Tester l'unique constructeur de *AElement*, l'opérateur de conversion en *string* et l'opérateur <<.

3. Mise en oeuvre de la classe générique GElement<T>

GElement<T> (pour Graphe-Element) est la classe de base dont héritent les classes *Sommet<T>* et *Arete<S,T>* (en fait, *Arete<S,T>* hérite de *GElement<S>*). *GElement<T>* hérite de *AElement*.
GElement<T> permet de factoriser l'information *v* présente à la fois dans les classes *Sommet<T>* et *Arete<S,T>*. *GElement<T>* est donc une classe dont le seul attribut est un attribut *v* (pour *valeur*) de type *T*.
Il faut donc :
Ecrire la classe *GElement<T>* ainsi définie. Pour simplifier, le membre *v* peut être public.
Munir *GElement<T>* d'un constructeur, d'un opérateur de conversion en *string* (fonction membre dont la signature est : ***operator*** *string () const*) et de l'opérateur << d'écriture sur un flux (cette dernière méthode est une fonction ordinaire, elle peut être *friend*, elle peut avantageusement se servir de l'opérateur de conversion en *string*).
Il est inutile d'écrire destructeur, constructeur de copie ou getters et setters.
Ecrire (dans un fichier *TestGrapheGeneral.cpp*) une fonction *main*() de test des fonctionnalités des classes *AElement GElement<T>*, *Sommet<T>*, *Arete<S,T>* et *Graphe<S,T>*. Tester l'unique constructeur de *GElement<T>*, l'opérateur de conversion en *string* et l'opérateur <<.

4. Mise en oeuvre de la classe générique Sommet<T>

La classe générique *Sommet<T>* représente un sommet dans un graphe.
Elle dérive de *GElement<T>* et contient un seul attribut, *degre*, qui est un nombre entier représentant le degré du sommet.
Ecrire la classe *Sommet<T>* ainsi définie. Pour simplifier, *degre* peut être public.
Munir *Sommet<T>* d'un constructeur, d'un opérateur de conversion en *string* (fonction membre dont la signature est : ***operator*** *string () const*) et de l'opérateur << d'écriture sur un flux (cette dernière méthode est une fonction ordinaire, elle peut être *friend*, elle peut avantageusement se servir de l'opérateur de conversion en string).
Par défaut, un sommet est créé isolé, donc de degré nul.
Nous faisons l'hypothèse que la classe *T* est munie d'un opérateur << d'écriture sur un flux.
Il est inutile d'écrire destructeur, constructeur de copie ou getters et setters. Tester constructeur, operateur string et opérateur << pour les classes *Sommet<double>* et *Sommet<string>*.

5. Mise en oeuvre de la classe générique Arete<S,T>

La classe générique *Arete<S,T>* représente une arête dans un graphe.
Elle dérive de *GElement<S>* et contient deux attributs :
debut : de type *Sommet<T>** représentant un pointeur sur l'extrémité initiale de l'arête
fin : de type *Sommet<T>** représentant un pointeur sur 1'extrémité finale de l'arête
Notons que, puisqu'elle hérite de *GElement<S>*, l'arête dispose aussi d'un attribut *v*, instance de la classe générique *S*, représentant l'information associée à l'arête.
Ecrire la classe *Arete<S,T>* ainsi définie. Pour simplifier, *debut* et *fin* peuvent être publics.
Munir *Arete<S,T>* des méthodes suivantes :

5.1 Constructeur

Ce constructeur peut avoir la signature suivante :

```
/**
met à jour le degré des sommets que cette nouvelle arête va connecter
*/
Arete( const int clef, const S & v,
Sommet<T> * debut, Sommet<T> * fin);
```

L'unique constructeur de *Arete<S,T>* ne fait pas de copie des arguments *debut* et *fin*.


Il modifie par contre ces deux mêmes arguments puisqu'il augmente leurs degrés respectifs !

5.2 Destructeur


Un destructeur est nécessaire puisque la destruction d'une arête entraîne une modification du degré des sommets que cette arête connectait.

5.3 Opérateur de conversion en *string*

Nous faisons l'hypothèse que la classe *S* est munie de l'opérateur << d'écriture sur un flux.
Il est suffisant, pour la conversion en *string* d'une arête, d'indiquer les clefs primaires des sommets *debut* et *fin*.

5.4 Opérateur << d'écriture sur un flux.

5.5 méthode estEgal(...)

dont la signature est :

```
/**
 * vérifie si *this s'appuie sur s1 et s2
 *
 * DONNEES : *this, s1, s2
 *
```

```
 * RESULTATS : true si *this s'appuie sur s1 et s2 c'est-à-dire si
(début == s1 et fin == s2) ou (début == s2 et fin == s1), false
sinon
 *
 * */
bool estEgal( const Sommet<T> * s1, const Sommet<T> * s2) const;
```

Cette méthode indique si *this relie s_1 et s_2.

Il est inutile d'écrire constructeur de copie, getters ou setters.

Tester constructeur, operateur string et opérateur << pour la classe *Arete<char,string>*.

6. Mise en oeuvre de la classe générique Graphe<S,T>

La classe générique *Graphe<S,T>* représente un graphe quelconque. *T* est la nature des informations associées aux sommets et *S* celle des informations associées aux arêtes.
Un graphe est défini par une liste de sommets, par une liste d'arêtes et par un générateur de clefs primaires.
La classe Graphe<S,T> est donc munie de trois attributs :
lSommets : de type *PElement< Sommet<T> >* *, qui contient la liste des sommets.
lAretes : de type *PElement<Arete<S,T>>* *, qui contient la liste des arêtes.
prochaineClef : de type *int* qui définit la clef primaire du prochain élément (sommet ou arête) qui sera inséré dans le graphe. *prochaineClef* est incrémenté à chaque insertion d'un élément.

6.1 Ecrire la définition de la classe Graphe <S,T>

Ecrire la définition de la classe avec ses attributs.

6.2 Munir la classe Graphe <S,T> du constructeur suivant :

```
Graphe();
```

qui crée un graphe vide. Initialise de façon cohérente les attributs *prochaineClef*, *lSommets* et *lAretes*.

6.3 Munir la classe Graphe<S,T> de la méthode suivante :

```
/**
 * crée un sommet isolé
 * met à jour prochaineClef
 * */
Sommet<T> * creeSommet(const T & info);
```

qui crée, dans le graphe, un sommet isolé muni de l'information *info*. Un pointeur sur le sommet créé est retourné.

6.4 Munir la classe Graphe<S,T> de la méthode suivante :

```
/**
 * crée une arête joignant les 2 sommets debut et fin
 *
 * met à jour les champs degre de debut et de fin
 *
 * met à jour prochaineClef
```

```
 *
 * */
Arete<S,T> * creeArete(const S & info, Sommet<T> * debut, Sommet<T> *
fin);
```

qui crée, dans le graphe, une arête reliant les sommets *debut* et *fin*. A l'appel, on suppose que *debut* et *fin* sont déjà contenus dans le graphe. L'arête créée contient l'information *info*. La méthode ne fait pas de copie de *debut* ou de *fin*. Les degrés de debut et de fin sont mis à jour par la méthode. Un pointeur sur l'arête créée est retourné.

6.5 Munir la classe Graphe<S,T> de la méthode suivante :

```
int nombreSommets() const;
```

qui renvoie le nombre de sommets contenus dans le graphe.

6.6 Munir la classe Graphe<S,T> de la méthode suivante :

```
int nombreAretes() const;
```

qui renvoie le nombre d'arêtes contenues dans le graphe.

6.7 Munir la classe Graphe<S,T> des opérateurs suivants :

opérateur de conversion en *string* et opérateur << d'écriture sur un flux.

6.8 Munir la classe Graphe<S,T> de la méthode suivante :

```
PElement< pair< Sommet<T> *, Arete<S,T>* > >  *
adjacences(const Sommet<T> * sommet) const;
```

qui construit, pour le sommet *sommet*, la liste de toutes les associations (voisin de *sommet*, arête incidente en *sommet*).
De cette liste, on peut donc déduire facilement, par exemple, la liste des voisins de *sommet* ou encore la liste des arêtes adjacentes à *sommet*.
Cette liste est essentielle au déroulement de A*.

6.9 Munir la classe Graphe<S,T> de la méthode suivante :

```
PElement< Arete<S,T> > *  aretesAdjacentes(const Sommet<T> * sommet)
const;
```

qui construit, pour le sommet *sommet*, la liste de toutes les arêtes incidentes en *sommet*.
Cette méthode est facilement déduite de la méthode *adjacences* (cf. paragraphe 6.8).

6.10 Munir la classe Graphe<S,T> de la méthode suivante :

```
PElement< Sommet<T> > *  voisins(const Sommet<T> * sommet) const;
```

qui construit, pour le sommet *sommet*, la liste de tous les sommets voisins en *sommet*.

Cette méthode est facilement déduite de la méthode *adjacences* (cf. paragraphe 6.8).
Rappel : Les voisins de *sommet* sont tous les sommets atteignables à partir de *sommet* en empruntant un chemin constitué d'une seule arête.

6.11 Munir la classe Graphe<S,T> de la méthode suivante :

```
/**
 * cherche l'arête s1 - s2 ou l'arête s2 - s1 si elle existe
 *
 * DONNEES : s1 et s2 deux sommets quelconques du graphe
 * RESULTATS : l'arête s'appuyant sur s1 et s2 si elle existe, NULL
sinon
 *
 * */
Arete<S,T> * getAreteParSommets( const Sommet<T> * s1, const
Sommet<T> * s2) const;
```

qui recherche dans le graphe la première arête qui relie s_1 et s_2.


Note : Ici, on doit aussi mettre en oeuvre un constructeur de copie, un destructeur et l'opérateur d'affectation =. Ces trois méthodes sont nécessaires à la garantie d'une gestion rigoureuse de la mémoire. C'est ici obligatoire car la classe comporte des membres alloués dynamiquement par ses soins.

6.12 Munir la classe Graphe <S,T> d'un constructeur de copie

Celui-ci est assez difficile à écrire. Il pourra être réalisé en dernier.

6.13 Munir la classe Graphe <S,T> de l'opérateur d'affectation :

```
const Graphe<S,T> & operator = ( const Graphe<S,T> & graphe);
```

Celui-ci étant assez difficile à écrire, il pourra être réalisé à la suite du constructeur de copie. Les deux méthodes étant assez similaires, elles aboutiront nécessairement à une factorsiation de code source pour éviter les redondances.

6.14 Munir la classe Graphe<S,T> d'un destructeur

Attention, sous peine de gros bugs à l'exécution, cette méthode doit impérativement être écrite seulement **après** avoir écrit (et mis en production) les constructeurs de copie et opérateur d'affectation ! Cette méthode aboutira à la mise en commun de code source avec l'opérateur d'affectation, attention : pas de copié-collé !

6.15 Tester les fonctionnalités de Graphe<S,T>

Tester les fonctionnalités de *Graphe<S,T>* **au fur et à mesure** qu'elles sont écrites.

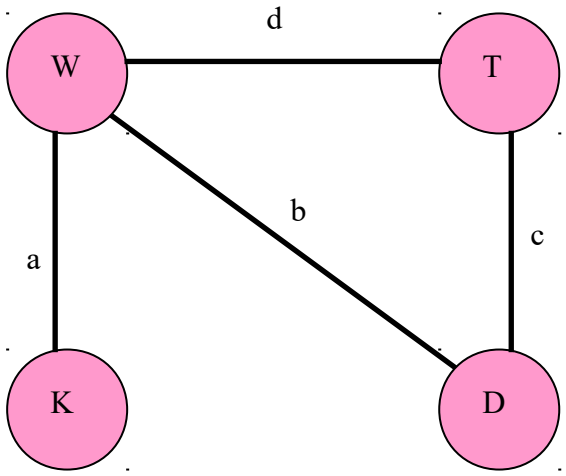

Créer un *Graphe<char, string>* à 4 sommets, dont un isolé, puis tester dans l'ordre le constructeur de graphe vide, l'opérateur <<, les méthodes *creeSommet()*, *creeArete()* puis *nombreSommets()*, *nombreAretes()*, *voisins()* puis enfin *aretesAdjacentes()*.

La fonction *main()* suivante pourra être utilisée pour valider les tests :

```
/*

Test des méthodes de base sur un graphe sauf les opérations de
dessin

L'info associée aux sommets est un string  (par exemple)
L'info associée aux arêtes est un char     (par exemple)

*/
#include <iostream>
#include <string>
#include <Graphe.h>

using namespace std;

int main()
{
char ch;
Graphe<char, string> g2;         // pour vérifier que
                                 // l'opérateur = et que le
                                 // destructeur fonctionnent bien

{
Graphe<char,string> g1;    // création à vide

Sommet<string> * s0, *s1, *s2, *s3;

//------------------ on insère des nouveaux sommets isolés
--------------------

s0 = g1.creeSommet("King's Landing");
s1 = g1.creeSommet("Winterfell");
s2 = g1.creeSommet("DragonStone");
s3 = g1.creeSommet("The wall");

//----------------- on connecte certains sommets ------------------

Arete<char,string> * a0, * a1, *a2, *a3;

a0 = g1.creeArete('a',s1,s0);
a1 = g1.creeArete('b',s2,s1);
a2 = g1.creeArete('c',s3,s2);
```

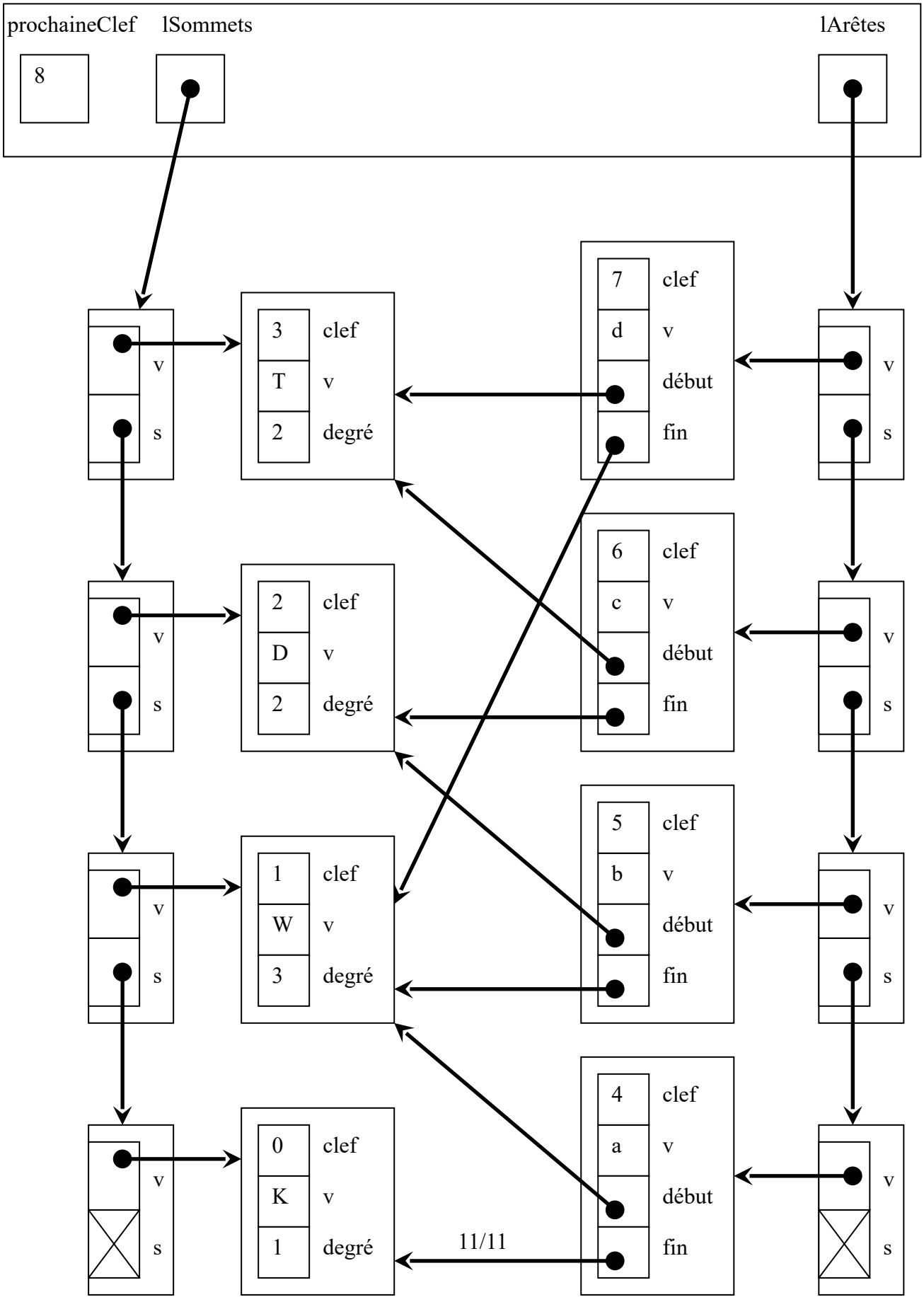
```
a3 = g1.creeArete('d',s3,s1);

//------------------ faire le dessin du graphe sur papier en notant
les noms et les degrés pour comprendre la suite ----------------


cout <<"le graphe créé g1 est :" << endl << g1 << endl; cin >> ch;

cout <<"le nombre de sommets de g1 est : " << g1.nombreSommets() <<
endl;
cout <<"le nombre d'arêtes de g1 est : " << g1.nombreAretes() <<
endl; cin >> ch;

PElement<Sommet<string>> * l0 = g1.voisins(s0);
cout << "la liste des voisins de s0 est : " << endl << l0 << endl;
cin >> ch;

PElement<Arete<char,string>> * adj0 = g1.aretesAdjacentes(s0);
cout << "la liste des arêtes adjacentes à s0 est : " << endl << adj0
<< endl; cin >> ch;

PElement<Sommet<string>> * l1 = g1.voisins(s1);
cout << "la liste des voisins de s1 est : " << endl << l1 << endl;

PElement<Arete<char,string>> * adj1 = g1.aretesAdjacentes(s1);
cout << "la liste des arêtes adjacentes à s1 est : " << endl << adj1
<< endl; cin >> ch;

Arete<char,string> * a = g1.getAreteParSommets(s1,s3);

cout <<"l'arête joignant s1 et s3 est : " << endl << *a << endl; cin
>> ch;

Graphe<char, string> g2;

g2 = g1;
}               // à la fin de ce bloc, le destructeur est appelé
                //  pour g1.
                // cela permet de vérifier que l'op = a fait une
                // vraie copie de g1

cout <<"le graphe créé g2 comme copie de g1 est :" << endl << g2 <<
endl; cin >> ch;


return 0;
}
```

Le graphe g_1 créé par la fonction *main()* a la forme suivante :



Le graphe g_1 créé dans l'exemple de la fontion *main()*ci-dessus

Et en mémoire centrale, avec la structure de graphe que nous avons définie, g_1 prend la forme suivante :


prochaineClef
lSommets
lArêtes
8
v
s
3
clef
T
v
2
degré
7
clef
d
v
début
fin
2
D
6
c
1
W
3
5
b
0
K
1
4
a
11/11