Structures de données Avancée

Cours et exercices
Filière SMI
2015-2016

Mustapha kchikech

Département de Mathématiques
et Informatique
Faculté Polydsciplinaire-Safi
Université Cadi Ayyad

جامعة القاضي عياض

FP-SAFI

Table des matières

Introduction

Ce polycopié présente les notions essentiels sur les structures de données avancées notamment les arbres binaires et les graphes.

L'objectif est de présenter des concepts fondamentaux de la programmation avancée en utilisant le langage C comme support du codage et mettre à la disposition des étudiants un support de cours qui pourrait leur permettre d'acquérir des compétences en programmation en faisant appel à des outils plus avancées comme les arbres binaires et les graphes.

Les définitions des arbres binaires, la façon avec laquelle sont implémentées et les différentes méthodes permettant de les manipuler sont présentées en chapitre 1. En suite les arbres AVL qui sont une version plus structurée et plus optimale des arbres binaires en la gestion des données sont présentés dans le chapitre 2. Le chapitre 3 et 4 sont consacrés respectivement sur les notions des arbres croissants et leur applications dans le tri des tableaux et les graphes et leurs importances dans la modélisation de nombreux problèmes d'aspect théorique ou pratique.

Par conséquent, ce cours offre aux étudiants la possibilité d'aborder et de réaliser des programmes assez avancée qui peuvent résoudre un grand nombre de problèmes de grande envergure.

Notons que ce document est un support pédagogique qui s'adresse principalement aux étudiants de la filière SMI semestre 4 de la Faculté Poly-disciplinaire de Safi, université cadi Ayyad inscrits en module structures de données. Il pourrait également servir comme support de cours aux étudiants d'autres filières.

Chapitre 1

Les arbres binaires

1.1 Introduction

— Un *arbre* est une structure de données particulière qui sert à stocker et manipuler des données dans des ensembles dynamiques.

— Sa particularité réside dans son concept d'organisation de données d'une manière hiérarchique.

— En effet, l'organisation des données consiste à relier les informations sur ces données par le biais des branches de l'arbre.

— Ceci permet un accès rapide aux informations stockées dans l'arbre et la possibilité d'ajouter et de supprimer, d'une manière également rapide, des éléments dans l'arbre.

— Par ailleurs, les structures de données arbres représentent l'un des concepts algorithmiques les plus importants de l'informatique.

— Ils sont utilisés dans plusieurs types d'applications informatiques. Exemple des utilisations les plus connus est la gestion des systèmes de bases de données, l'organisation des systèmes de fichiers d'un système d'exploitation ou combinatoire des mots, codage, etc.

1.2 Arbres binaires

1.2.1 Présentation générale

— On peut présenté un arbre comme étant un nœud ou une suite de sous-arbres.

— Un arbre peut être un arbres enraciné (avec racine) ou arbre non enraciné (sans racine).

— Il existe plusieurs type d'arbres mais, vu leur intérêt en informatique, nous allons étudier dans ce chapitre seulement les arbres binaires.

— Dans un arbre binaire, un nœud désigne l'élément de l'information et les branches vers d'autres nœuds.

— Les nœuds sont reliés les uns aux autres par des relations d'ordre ou de hiérarchie.

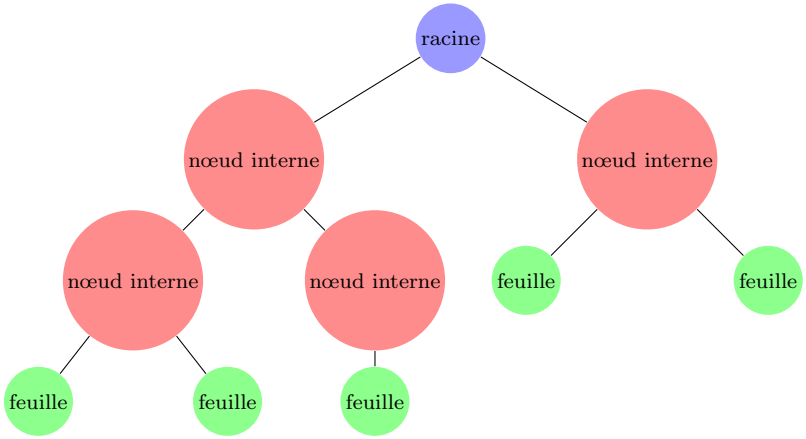
— Ainsi, chaque nœud pourrait avoir un père (nœud qui lui est supérieur dans la hiérarchie), un ou deux fils : fils gauche et fils droit.

— La racine est le nœud qui n'a pas de père, c'est donc la racine de l'arbre. La racine est le nœud qui impose un sens de parcours de l'arbre.

— Un nœud qui n'a pas de fils est appelé une feuille.

— Le reste des nœuds de l'arbre seront alors appelés un nœuds internes.

— Exemple :



1.2.1.1 Opérations sur les arbres binaires

— Dans un arbre binaire, on peut effectuer plusieurs opérations par exemple :

1. *initialiser* un arbre binaire,
2. *tester* si un arbre est vide,
3. *créer* l'arbre gauche et l'arbre droit d'un arbre binaire,
4. *parcourir* un arbre,
5. *ajouter* un élément à un arbre,
6. *supprimer* un élément d'un arbre,
7. *récupérer* un élément d'un arbre,
8. *vider* un arbre.
9. ...

1.2.2 Implantation

1.2.2.1 Déclaration d'un arbre binaire

— Comme le cas des listes chaînée, les piles et les files, le langage C permet de réaliser des arbres binaires à l'aide des structures.

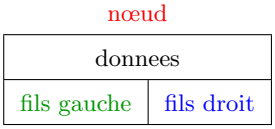
— En effet, chaque noœud de l'arbre est un objet constitué :

  — d'un champ de données,

  — d'un pointeur vers le fils gauche (sous-arbre gauche).

  — d'un pointeur vers le fils droit (sous-arbre droit).

— La déclaration correspondante est la suivante, où l'on suppose ici que les valeurs stockées dans l'arbre sont de type entier.



— Code en C :

```
typedef int element;
struct noeud
{
 element valeur;
 struct noeud * filsG;
```

```
 struct noeud * filsD;
};
typedef struct noeud noeud, * arbreB;
```

1.2.2.2 Création et initialisation d'un arbre binaire

— Pour créer un arbre binaire, on doit l'initialiser à `NULL`, cela va nous permettre d'allouer le premier nœud.

— L'initialisation d'un arbre binaire, ou la création d'un arbre binaire, est réalisée par la fonction suivante :

```
arbreB arbreBVide( )
{
 return NULL;
}
```

— Pour tester si un arbre binaire est vide, on utilise la fonction suivante :

```
int testerArbreBVide(arbreB a)
{
 return a==NULL;
}
```

1.2.2.3 Création d'un arbre binaire

— Pour créer un arbre binaire, Il faut tout d'abord créer un nœud, ensuite on construit, d'une manière récursive, les sous-arbres gauche et droit.

— Pour ceci, nous allons définir deux fonctions qui permettent de donner le fils gauche, le fils droit d'un nœud et une fonction qui permet de créer un nœud

— Fonctions qui donnent le fils gauche et le fils droit d'un nœud :

  — Fils gauche :

```
arbreB filsGauche(arbreB a)
{
 if(a==NULL)
   return NULL;
 return a->filsG;
}
```

— Fils droit :

```
arbreB filsDroit(arbreB a)
{
 if(a==NULL)
   return NULL;
 return a->filsD;
}
```

— Fonction qui permet de créer un nœud :

```
arbreB creeNoeud(element val,arbreB fg, arbreB fd)
{
 arbreB a=malloc(sizeof(noeud));
 a->valeur=val;
 a->filsG=fg;
 a->filsD=fd;
 return a;
}
```

1.2.2.4 Ajout d'un élément dans un arbre binaire

— La structure d'un arbre binaire, rend l'opération d'Ajout d'un élément un peu délicat.

— En effet, il existe différentes façons d'insérer un élément dans un arbre binaire :

  — On insère des éléments dès que l'on peut.

  — On insère des éléments de façon à obtenir un arbre qui se rapproche le plus possible d'un arbre complet (chaque nœud possède 0 ou 2 fils.),

  — On insère de façon à avoir une certaine logique dans l'arbre binaire à construire.

— La première méthode d'ajout est la plus simple, dès que l'on trouve un nœud qui a un fils vide, on y met le sous-arbre que l'on veut insérer.

— Mais cette méthode présente un inconvénient. En effet, l'arbre construit est un arbre très particulier connu sous le nom d'arbre peigne (arbre où tous les fils droit (ou gauche) des nœuds internes sont des feuilles. Voici le code en C de cette méthode.

— Fonction qui ajoute des nœuds côté gauche :

```
arbreB ajouterGElementArbreB(arbreB a, element val)
{
 if(a==NULL)
    a=creeNoeud(val,NULL,NULL);
 else if(filsGauche(a)==NULL)
    a->filsG=creeNoeud(val,NULL,NULL);
 else if(filsDroit(a)==NULL)
    a->filsD=creeNoeud(val,NULL,NULL);
 else
    a->filsG=ajouterGElementArbreB(filsGauche(a), val);
 return a;
}
```

— Fonction qui ajoute des nœuds côté droit :

```
arbreB ajouterDElementArbreB(arbreB a, element val)
{
 if(a==NULL)
   a=creeNoeud(val,NULL,NULL);
 else if(filsGauche(a)==NULL)
   a->filsG=creeNoeud(val,NULL,NULL);
 else if(filsDroit(a)==NULL)
   a->filsD=creeNoeud(val,NULL,NULL);
 else
   a->filsD=ajouterDElementArbreB(filsDroit(a), val);
 return a;
}
```

1.2.2.5 Arbre binaire de recherche

— On remarque la création d'arbres avec la première méthode n'est pas pratique.

— On présente une autre façon de créer des arbres binaires en se basant sur une logique. Les arbres binaires de recherche.

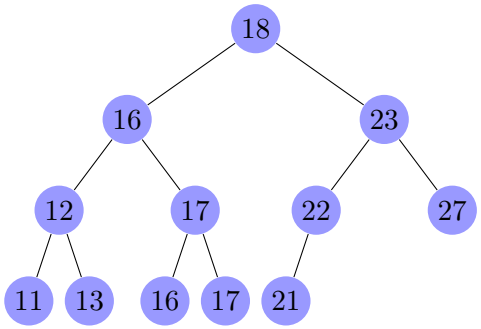
— Ce type d'arbre, il y a une cohérence entre les nœuds, c'est à dire que la hierarchie des nœuds respecte une règle.

— Pour un arbre binaire de recherche contenant des entiers, les valeurs des nœuds

des sous-arbres gauche sont inférieures à la racine de ce nœud,

— et les valeurs des sous-arbres droit sont supérieurs à cette racine.

— Exemple :



1.2.2.6 Ajout d'un élément dans arbre binaire de recherche

Fonction qui ajoute des nœuds dans arbre binaire de recherche :

```
arbreB AjoutarbreRecherche(arbreB a, element val)
{
 if(a==NULL)
   a=creeNoeud(val,NULL,NULL);
 else if(a->valeur >= val)
   a->filsG=AjoutarbreRecherche(filsGauche(a), val);
 else
   a->filsD=AjoutarbreRecherche(filsDroit(a), val);
 return a;
}
```

1.2.2.7 Exercices d'application

Écrire un programme en C qui permet de réaliser les fonctions suivantes :

— Fonction qui teste si un nœuds est une feuille,

— Fonction qui teste si un nœuds est un nœuds interne,

— Fonction qui calcule le nombre de nœuds d'un arbre binaire,

— Fonction qui calcule le nombre de nœuds internes d'un arbre binaire,

— Fonction qui calcule le nombre de feuilles d'un arbre binaire,

— Fonction qui calcule la hauteur d'un arbre binaire,

1.2.3 Parcours d'un arbre

— Pour pouvoir traiter les données stockées dans un arbre binaire, il faut parvenir à atteindre les nœuds contenants ces données.

— Ceci demande des méthodes ou des algorithmes qui permettent d'explorer un arbre dans le but de visiter tous ses nœuds.

— Il existe deux méthodes de parcours d'un arbre binaire :

— le parcours en profondeur :

— Le parcours en largeur :

1.2.3.1 Parcours en profondeur

— Parcours en profondeur : permet d'explorer l'arbre en explorant jusqu'au bout une branche pour passer à la suivante.

— Dans le parcours en profondeur, on distingue trois types de parcours :

— parcours préfixe : On explore la racine de l'arbre, on parcourt tout le sous-arbre de gauche, une fois qu'il n'y a plus de sous-arbre gauche on parcourt les éléments du sous-arbre droit. Parcours RGD (Racine-Gauche-Droit)

— parcours infixe : explore la racine après avoir traité le sous-arbre gauche, après traitement de la racine, on traite le sous-arbre droit. Parcours GRD (Gauche-Racine-Droit).

— parcours suffixe (ou postfixe) : explore le sous-arbre gauche, le sous-arbre droit puis la racine. Parcours GDR (Gauche-Droit-Racine).

1.2.3.2 Parcours en profondeur : Parcours préfixe

Principe : pour chaque nœud, on visite la racine puis le fils gauche et en dernier le fils droit.

```
void prcoursPrefixe(arbreB a)
{
 if(a!=NULL)
  {
   printf("%d ",a->valeur);//ici visite (traite)
```

```
    prcoursPrefixe(filsGauche(a));
    prcoursPrefixe(filsDroit(a));
  }
}
```

1.2.3.3 Parcours en profondeur : Parcours Infixe

Principe : pour chaque nœud, on visite le fils gauche puis la racine et en dernier le fils droit.

```
void prcoursInfixe(arbreB a)
{
 if(a!=NULL)
  {
   prcoursPrefixe(filsGauche(a));
   printf("%d ",a->valeur);//ici visite (traite)
   prcoursPrefixe(filsDroit(a));
  }
}
```

1.2.3.4 Parcours en profondeur : Parcours suffixe (postefixe)

Principe : pour chaque nœud, on visite le fils gauche puis le fils droit et en dernier la racine.

```
void prcoursSuffixe(arbreB a)
{
 if(a!=NULL)
  {
   prcoursPrefixe(filsGauche(a));
   prcoursPrefixe(filsDroit(a));
   printf("%d ",a->valeur);//ici visite (traite)
  }
}
```

1.2.3.5 Parcours en largeur

- Le parcours en largeur : permet d'explorer l'arbre niveau par niveau.
- L'algorithme du parcours en largeur consiste à utiliser une file pour garder en mémoire les fils du nœuds visité dans chaque niveau pour les traiter après.

```
void parcoursLargeur(arbreB a)
{
 File f=NULL;
 arbreB atmp;//noeud temporaire
 if(a!=NULL)
  {
   f=enfiler(a,f);
   while( f!=NULL)
    {
     atmp=defiler(&f);
     printf("%d ",atmp->valeur);//ici visite (traite)
     if(filsGauche(atmp)!=NULL)
        f=enfiler(atmp->filsG,f);//ici mettre en memoire le fils gauche
     if(filsDroit(atmp)!=NULL)
        f=enfiler(atmp->filsD,f);//ici mettre en memoire le fils droit
    }
  }
}
```

1.2.4 Recherche dans un arbre

— La recherche d'une valeur stockée dans un arbre est une opération très courantes sur les arbres binaires.

— On distingue principalement deux méthodes de recherche liées au type de l'arbre binaire considéré.

  — Si l'arbre est quelconque.

  — Si l'arbre est un arbre de recherche.

— L'objectif de la recherche est de déterminer si la valeur existe dans l'arbre.

1.2.4.1 Recherche dans un arbre binaire quelconque

— Principe :
La fonction commence sa recherche à la racine. Pour chaque nœud visité, si la valeur recherchée est trouvée, la recherche est terminée. Sinon, la recherche continue dans le sous-arbre gauche et le sous-arbre droit.

```
int rechercherEltArbre(arbreB a,element val)
{
 if(a==NULL)
   return 0;
 if(a->valeur==val)
   return 1;
 return rechercherEltArbre(filsGauche(a), val)
        || rechercherEltArbre(filsDroit(a), val);
}
```

1.2.4.2 Recherche dans un arbre binaire de recherche

— Contrairement à la recherche dans un arbre quelconque où il faut parcourir quasiment tous les nœuds de l'arbre pour déterminer si l'élément existe,

— la recherche dans un arbre binaire de recherche permet de parcourir seulement une branche de l'arbre.

— Principe : La fonction commence sa recherche à la racine. Pour chaque nœud visité, si la valeur recherchée est trouvée, la recherche est terminée. Sinon, la recherche continue dans le sous-arbre gauche ou le sous-arbre droit.

```
int rechercherEltArbreRech(arbreB a,element val)
{
 if(a==NULL)
  return 0;
 if(a->valeur==val)
  return 1;
 if(a->valeur>=val)
  return rechercherEltArbreRech(filsGauche(a), val);
 return rechercherEltArbreRech(filsDroit(a), val)
}
```

1.2.5 Suppression d'un élément dans arbre binaire

— La suppression d'une valeur (ou un nœud) dans un arbre est une opération plus complexe.

- En effet, supprimer un élément dans un arbre revient à détruire le nœud contenant cet élément.
- Cependant, un nœud ayant des fils s'il est supprimé, entraîne une réorganisation de l'arbre.
- Autrement, que faire alors des sous-arbres du nœud à supprimer ?
- La réponse dépend du type de l'arbre traité.
  - Si l'arbre considéré est un arbre binaire de recherche, on peut supprimer un nœud tout en gardant l'aspect de cet arbre, c-à-d garder l'organisation de l'arbre selon le concept des arbres binaires de recherche.
  - Si l'arbre considéré est un arbre binaire quelconque, on peut supprimer le nœud et ses sous-arbres ou supprimer seulement le nœud.
- Dans la suite, nous allons présenter une fonction qui permet de supprimer un nœud dans un arbre binaire de recherche sans supprimer ses fils.
- Une autre fonction qui permet de détruire complètement un nœud et ses descendants.

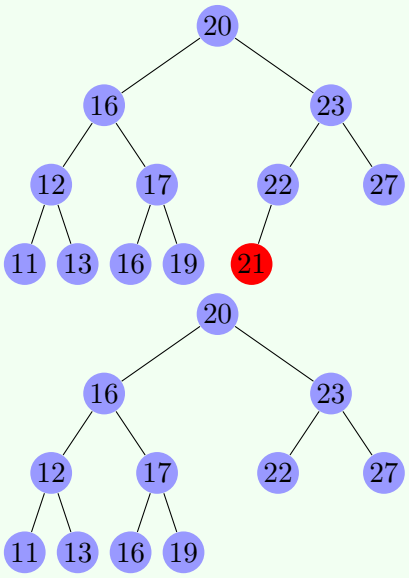
1.2.5.1 Suppression d'un élément dans un arbre binaire de recherche

- Principe : Si n est le nœud qui contient l'élément à supprimer, alors trois cas sont envisageables :
  - Si n est une feuille, alors on le supprime.
  - Si n est un nœud ayant un seul fils, alors on supprime n et on le remplace par ce fils.
  - Si n est un nœud ayant deux fils (gauche et droit), alors on cherche dans le sous-arbre gauche la feuille x qui porte le plus grand élément. Ensuite, on remplace le contenu de n par le contenu de x, et on supprime x.
- Trois fonctions coopèrent pour réaliser cette tâche :
  - une fonction permet de supprimer la racine d'un nœud.
  - une fonction permet de déterminer le maximum des éléments d'un arbre binaire de recherche.

— une fonction permet de supprimer un élément dans un arbre binaire de recherche.

Exemple 1.

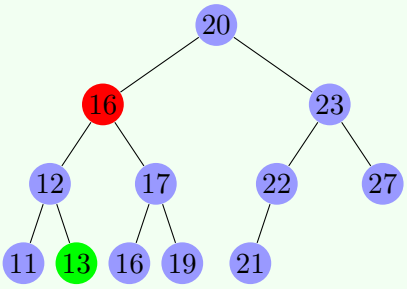
Suppression d'une feuille :



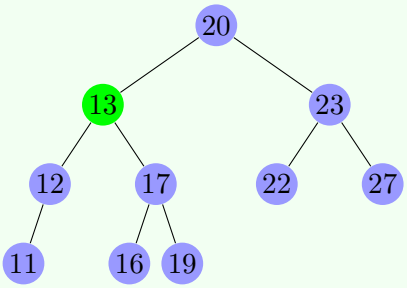
La suppression de 21 donne

Exemple 2.

Suppression d'un nœud à deux fils :



La suppression de 16 donne



Code en C :

— Maximum d'un arbre binaire de recherche :

```
arbreB MaxArbreRech(arbreB a)
{
 if(a->filsD == NULL)
   return a;
  return MaxArbreRech(a->filsD);
}
```

— Supprimer la racine d'un nœud :

```
arbreB supprimerRacine(arbreB a)
{
 arbreB tmp;
 arbreB supprimerElt((int v, arbreB a);
 if(a->filsG == NULL)
   return a->filsD;
```

```
 if(a->filsD == NULL)
   return a->filsG;
 tmp = MaxArbreRech(a->filsG);
 a->valeur= tmp->valeur;
 a->filsG = supprimerElt((tmp->valeur, a->filsG);
 return a;
}
```

— Supprimer la valeur d'un nœud :

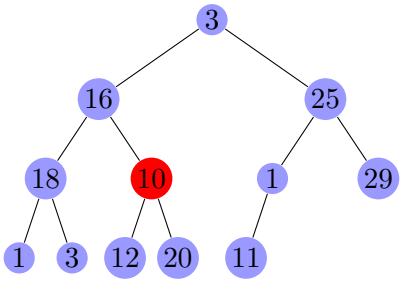
```
arbreB supprimerElt(int v, arbreB a)
{
 if(a==NULL)
   return a;
 if(v==a->valeur)
   return supprimerRacine(a);
 if(v<a->valeur)
   a->filsG = supprimerElt(v, a->filsG);
 else
   a->filsD = supprimerElt(v, a->filsD);
 return a;
}
```

1.2.5.2 Suppression d'un élément dans un arbre binaire quelconque

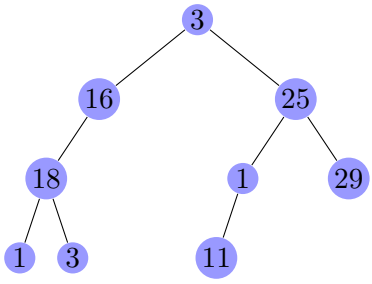
— Principe : Si n est le nœud qui contient l'élément à supprimer, alors on supprime complètement l'arbre de racine n.

— L'idée est de supprimer les feuilles une par unes. On réitère l'opération autant de fois qu'il y a de feuilles.

— Exemple :



La suppression de 10 donne



Code en C :

— Vider un arbre :

```
arbreB viderArbreB(arbreB a)
{
 if(a!= NULL)
  {
   a->filsG=viderArbreB(a->filsG);
   a->filsD=viderArbreB(a->filsD);
   free(a);
   a=NULL;
   return a;
  }
  return NULL;
}
```

— Supprimer un nœud :

```
arbreB supprimerNoeudQcq(int v, arbreB a)
{ arbreB tmp;
 if(a== NULL)
   return a;
 if(v == a->valeur)
  {tmp=a;
   return viderArbreB(tmp);
  }
 if(rechercherArbre(filsGauche(a), v))
   a->filsG = supprimerNoeudQcq(v, a->filsG);
 else  a->filsD = supprimerNoeudQcq(v, a->filsD);
 return a;
}
```

Chapitre 2

Arbres binaires équilibrés AVL

2.1 Motivation et Présentation

— Nous avons vu que les opérations d'insertion ou de suppression des éléments dans un arbre binaire de recherche peut donner des arbres totalement déséquilibrés,

— autrement, la différence entre la hauteur des sous-arbres est assez importante. Il peut y avoir des arbres dont la profondeur est proche leur taille.

— Ceci peut influencer le coût en temps des traitements sur ce type d'arbre.

— Pour éviter ce genre d'arbres, on peut chercher à garder un arbre binaire de recherche sous une forme équilibrée après chaque opération d'insertion ou de suppression.

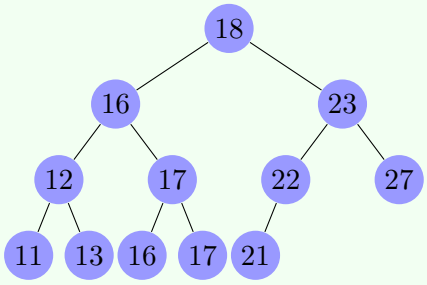
— En algorithmique, il existe plusieurs types d'arbres dits équilibrés les arbres AVL, les arbres rouge et noir, les arbres a-b.

Dans la suite, nous traitons le cas des arbres AVL (notion introduite en 1962 par deux russes Adelson-Velskii et Landis).

Définition 1.

Un arbre binaire de recherche est un arbre AVL si, pour tout nœud de l'arbre, les hauteurs des sous-arbres gauche et droit diffèrent d'au plus 1.

Exemple 3.



2.2 Opérations sur les arbres AVL

— L'insertion (ou la suppression) dans un arbre AVL se fait comme dans un arbre de recherche sauf qu'il faut maintenir l'équilibre de l'arbre.

— Pour cela, lors de l'insertion (ou suppression), s'il peut y avoir un déséquilibre trop important (≥ 2) entre les deux sous-arbres du nœud traité il faut créer un équilibre par des rotations (gauche ou droite).

— Les opérations de rotation transforment la configuration des sous-arbres de gauche ou de droit sans modifier l'ordre symétrique des nœuds (l'ordre dans le parcours infixe).

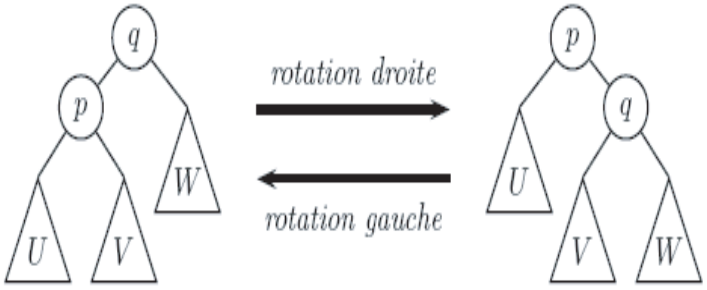
— La rotation préserve la propriété d'arbre binaire de recherche, en revanche elle peut diminuer la hauteur du sous-arbre.

— Rotation : Les rotations sont illustrées par la figure suivante :

— Équilibrer : si la hauteur des sous-arbres droit et gauche diffère d'au plus de 1, effectuer une rotation gauche ou droite ou les deux selon la situation.

— Insertion : on ajoute un élément à l'arbre AVL d'une manière similaire à un arbre binaire de recherche sauf qu'il faut préserver l'équilibre à chaque insertion.

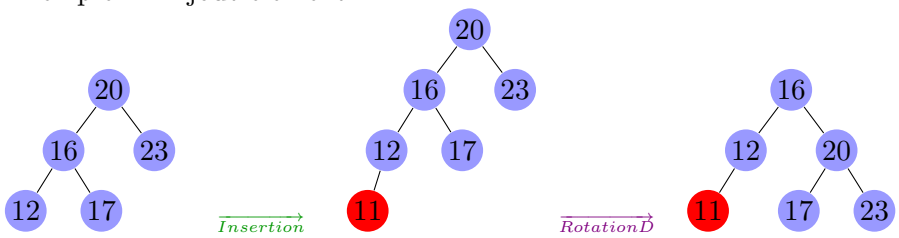
— Suppression : on supprime un élément de l'arbre AVL d'une manière similaire à un



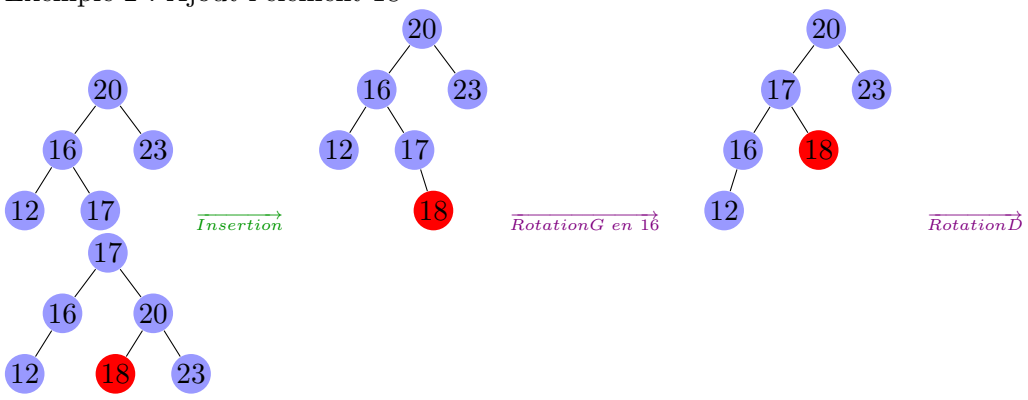
arbre binaire de recherche sauf qu'il faut préserver l'équilibre à chaque suppression.

2.2.1 Exemple d'ajout d'un élément dans un AVL

— Exemple 1 : Ajout élément 11

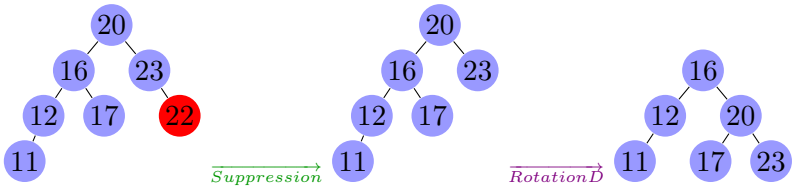


— Exemple 2 : Ajout l'élément 18

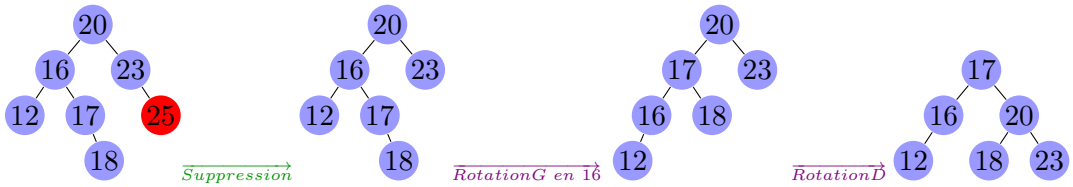


2.2.2 Exemple de suppression d'un élément dans un AVL

Exemple 1 : Suppression l'élément 22



Exemple 2 : Suppression de l'élément 25



2.3 Implantation d'un arbre AVL en C

Pour implanter en langage C les opérations d'insertion et de suppression des éléments dans un arbres AVL, il faut écrire des fonctions qui permettent

1. d'effectuer une rotation gauche sur un arbre binaire de recherche,

```
arbreB rotationG(arbreB a)
{
  arbreB b;
  b=a->filsD;
  a->filsD=b->filsG;
  b->filsG=a;
  return b;
}
```

2. d'effectuer une rotation droite sur un arbre binaire de recherche,

```
arbreB rotationD(arbreB a)
{
  arbreB b;
  b=a->filsG;
```

```
  a->filsG=b->filsD;
  b->filsD=a;
  return b;
}
```

3. d'équilibrer un arbre binaire de recherche,
   Fonction qui permet de calculer la différence de hauteur entre sous-arbre gauche et sous-arbre droit :

```
int diffHaut(arbreB a)
{
 return haut(a->filsD)-haut(a->filsG);
}
```

   Fonction d'équilibre :

```
arbreB equilibrerAVL(arbreB a)
{
 if(diffHaut(a)== 2)
    if(diffHaut(a->filsD) >= 0)
       return rotationG(a);
    else
 {
   a->filsD=rotationD(a->filsD);
          return rotationG(a);
 }
 else if (diffHaut(a)== -2)
        if (diffHaut(a->filsG)<= 0)
           return rotationD(a);
        else
          {
    a->filsG=rotationG(a->filsG);
           return rotationD(a);
          }
else return a;
}
```

4. D'ajouter un élément dans un arbre binaire AVL,

```
arbreB ajoutAVL(arbreB a,int val)
{
  if(a==NULL)
    a=creeNoeud(val,NULL,NULL);
```

```
  else if(a->valeur >= val)
    a->filsG=ajoutAVL(a->filsG , val);
      else
    a->filsD=ajoutAVL(a->filsD , val);
     a=equilibrerAVL(a) ;
  return a;
}
```

5. De supprimer un élément dans un arbre binaire AVL. Le code C est en exercice.

2.4 Exercices

Exercice 1 : Arbre binaire

Cet exercice a pour but d'implementer un arbre binaire en langage C. Pour ceci, il est demandé d'écrire un programme complet et structuré, utilisant des fonctions et des fichiers d'en-tête (.h), permettant aux utilisateurs de manipuler un arbre binaire. Les tâches demandées sont les suivantes :

1. Définir une structure représentant un arbre binaire.
2. Initialiser un arbre binaire.
3. Retourner les sous-arbres gauche et droit d'un arbre binaire.
4. Ajouter, dès que l'on peut, un nœud à un arbre binaire.
5. Créer un arbre binaire, en ajoutant des éléments :
   — par l'utilisateur (saisie au clavier),
   — ou par le programme (saisie aléatoire).
6. Afficher les éléments d'un arbre binaire en utilisant les méthodes de parcours d'un arbre.
   — Parcours en largeur,
   — Parcours préfixe,
   — Parcours infixe,
   — Parcours suffixe (postfixe),

7. Donner le nombre de nœuds d'un arbre binaire.
8. Tester si un nœud est une feuille ou un nœud interne.
9. Donner le nombre de feuilles et le nombre de nœuds internes d'un arbre binaire.
10. Calculer la hauteur d'un arbre binaire.
11. Rechercher un élément dans un arbre binaire.
12. Supprimer un nœud dans un arbre binaire.
13. Vider un arbre binaire.

Exercice 2 : Arbre binaire de Recherche

Écrire des fonctions qui permettent

1. d'insérer un élément dans un arbre binaire de recherche,
2. de créer (au clavier et aléatoirement) un arbre binaire de recherche,
3. de rechercher un élément dans un arbre binaire de recherche,
4. de supprimer un élément dans un arbre binaire de recherche,
5. de tester si un arbre binaire est un arbre de recherche,
6. Exemple : créer un arbre binaire de recherche contenant des chaînes de caractères de longueur au plus 4.

Exercice 3 : Arbres binaires équilibrés : arbres AVL

Cet exercice a pour but d'écrire un programme complet en C permettant d'implementer les arbres binaires équilibrés les arbres AVL.

Pour ceci, écrire des fonctions qui permettent

1. d'effectuer une rotation gauche sur un arbre binaire,
2. d'effectuer une rotation droite sur un arbre binaire,
3. d'équilibrer un arbre binaire,
4. d'ajouter un élément dans un arbre binaire AVL,
5. de créer (au clavier et aléatoirement) un arbre binaire AVL,

6. de supprimer un élément dans un arbre AVL,
7. de tester si un arbre binaire est un arbre AVL ?
8. Exemple : créer un arbre binaire AVL contenant des étudiants.

Chapitre 3

Arbres binaires croissants

3.1 Présentation

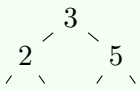
— On présente dans cette partie la structure d'arbre croissant, une structure de données pour réaliser des files de priorité.

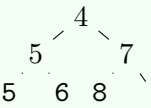
— En effet, cette structure permet d'insérer un élément, de supprimer le minimum, de trouver le minimum et de trier les éléments d'un ensemble de données.

— On dit qu'un arbre binaire a est un arbre binaire croissant (ABC) si, soit a est vide, soit les fils gauches ou les fils droits de a sont eux-même deux arbres croissants et la valeur de la racine est inférieur ou égal à tous les éléments du fils gauche et du fils droit.

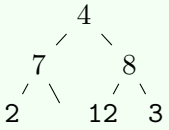
Exemple 4.

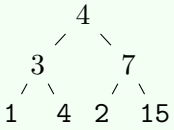
— Les deux arbres suivants sont croissants :





— Les deux arbres suivants ne sont pas croissants :





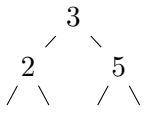
3.2 Notations

Dans cette partie, un arbre binaire est

— soit l'arbre vide, noté `V`,

— soit un nœud constitué d'un sous-arbre gauche, noté `fG`, d'une valeur (exemple un entier), noté `val` et d'un sous-arbre droit, noté `fD`, ce nœud est noté `N(fG,val,fD)`.

— Exemple : Par la suite, on peut dessiner l'arbre `N(N(V,2,V),3,N(V,5,V))` sous la forme



3.3 Opérations sur les arbres binaires croissants

Dans un arbre binaire croissant (ABC), on peut effectuer plusieurs opérations en particulier :

— Fusion de deux ABC.

— Insertion d'un élément.

— Suppression du plus petit élément.

3.3.1 Fusion de deux arbres binaires croissants

L'opération de fusion de deux arbres croissants. a_1 et a_2, noté $fusion(a_1, a_2)$, est définie récursivement de la manière suivante :

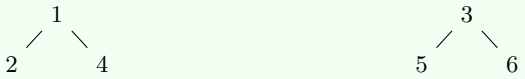
1. Si le sous-arbre droit est vide alors
   fusion(a_1,V)=a_1
2. Si le sous-arbre gauche est vide alors
   fusion(V,a_2)=a_2
3. Si $val_1 \leqslant val_2$ alors
   fusion(N(fG_1,val_1,fD_1),N(fG_2,val_2,fD_2))=N(fusion(fD_1,N(fG_2,val_2,fD_2)),val_1,fG_1)
4. Si $val_2 \leqslant val_1$ alors
   fusion(N(fG_1,val_1,fD_1),N(fG_2,val_2,fD_2))=N(fusion(fD_2,N(fG_1,val_1,fD_1)),val_2,fG_2)

Code en langage C

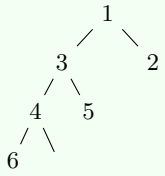
```
arbreB fusionABC(arbreB a, arbreB b)
{
 arbreB tmp;
 if(a==NULL)
   return b;
 if(b==NULL)
   return a;
 if(a->valeur<=b->valeur)
  {
   tmp=a->filsD;
   a->filsD=a->filsG;
   a->filsG=fusionABC(tmp,b);
   return a;
  }
 else
  {
   tmp=b->filsD;
   b->filsD=b->filsG;
   b->filsG=fusionABC(tmp,a);
   return b;
  }
}
```

Exemple 5.

La fusion de ces deux arbres croissants



donne l'arbre croissant suivant :



3.3.2 Ajout d'un élément dans un ABC

— L'ajout d'un élément dans un arbre binaire croissant consiste à fusionner l'arbre binaire croissant donné avec le nœud contenant l'élément à ajouter.

Code en langage C

```
arbreB creeNoeud(int val,arbreB fg, arbreB fd)
{
 arbreB a;
 a=malloc(sizeof(noeud));
 a->valeur=val;
 a->filsG=fg;
 a->filsD=fd;
 return a;
}

arbreB ajoutABC(int v,arbreB a)
{
 arbreB b;
 b=creeNoeud(v,NULL,NULL);
 a=fusionABC(a,b);
 return a;
}
```

— La fonction `creeNoeud(element v,arbreB fg,arbreB fd)` permet de créer un nœud (fonction définie en chapitre 1).

— La fonction `fusionABC(arbreB a,arbreB b)` permet de fusionner deux ABC.

3.3.3 Fournir le petit élément d'un ABC

— La structure d'un arbre binaire croissant permet de fournir le plus petit élément d'un ensemble de données stockées dans cet arbre. En effet, la valeur de la racine est le minimum des éléments de cet arbre.

Code en langage C

```
int minABC(arbreB a)
{
 return a->valeur;
}
```

3.3.4 Suppression du plus petit élément d'un ABC

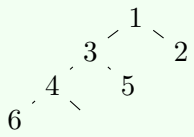
— La suppression du plus petit élément d'un arbre binaire croissant consiste à fusionner le sous-arbre gauche et le sous-arbre droit de l'arbre binaire croissant donné.

Code en langage C

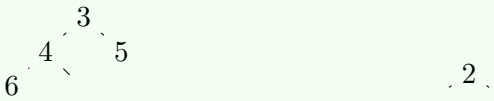
```
arbreB supprimMin(arbreB a)
{arbreB b;
 b=fusionABC(a->filsG,a->filsD);
 return b;
}
```

Exemple 6.

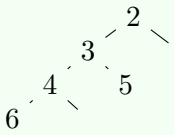
Suppression du minimum : La suppression du minimum de l'arbres croissants suivant



donne la fusion des deux sous-arbres suivants :



qui donne l'arbre croissant suivant :



3.3.5 Application : Tri de tableau en $\mathcal{O}(nlog(n))$

— La 1ère fonction permet de copier un tableau de n entiers dans un arbre binaire croissant.

— En utilisant la structure d'arbre binaire croissant, la 2ème fonction permet de trier un tableau de n entiers, dans l'ordre croissant, en temps $\mathcal{O}(nlog(n))$.

Code en langage C

```
arbreB TabToABC(int *t,int n){
 arbreB a=NULL,b;
 int i;
 for(i=0;i<n;i++){
    b=creeNoeud(t[i],NULL,NULL);
    a=fusionABC(a,b);
  }return a;
}
```

```
void triTab(int *t,int n)
{
 int i=0;
 arbreB a;
 a=TabToABC(t,n);
 while(a!=NULL)
 {
  t[i++]=a->valeur;
  a=supprimMin(a);
 }
}
```

Chapitre 4

Les Graphes

4.1 Présentation et définitions

Un graphe est une structure combinatoire permettant de représenter une structure d'un ensemble complexe en exprimant les relations entres ses éléments. En particulier, il permet de représenter de nombreuses situations rencontrées dans des applications faisant intervenir des mathématiques discrètes et nécessitant une solution informatique comme les circuits électriques, les réseaux de transport (ferrés, routiers, aériens), des réseaux d'ordinateurs, ou ordonnancement d'un ensemble de tâches,...

La théorie des graphes est une méthode de pensée, un moyen de modélisation qui permet l'étude d'une grande variété de problèmes : des problèmes ayant un aspect algorithmique, des problèmes d'optimisation combinatoire etc.

4.1.1 Définition d'un graphe

Définition 2.

Un graphe $G = (V, E)$ désigne le couple constitué par un ensemble fini de *sommets* $V = \{v_1, v_2, ..., v_n\}$, et par un ensemble $E \subset V \times V$.
Si G est un graphe non-orienté, les éléments de E sont appelés arêtes, sinon G est orienté et les éléments de E sont appelés arcs.
Le nombre de sommets du graphe G est appelé ordre de G, on le note soit $ordre(G)$ ou soit $|V|$.

4.1.2 Représentation graphique

dans le cas d'un graphe orienté, la relation entre deux sommets est représentée à l'aide d'une flèche (arc) entre ceux-ci. Dans le cas d'un graphe non orienté, deux sommets qui forment une arête sont reliés par un trait. Notons $G_a = (V_a, E_a)$ et $G_b = (V_b, E_b)$

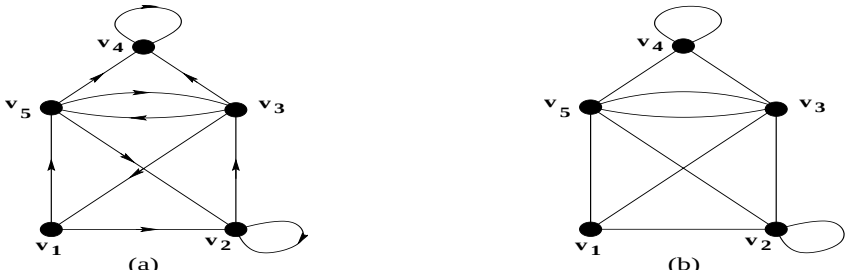


FIGURE 4.1 – (a) : graphe orienté ; (b) : le même graphe non orienté.

respectivement les graphes de la Figure (a) et la Figure (b).

— $V_a = V_b = \{v_1, v_2, v_3, v_4, v_5\}$, $|V_a| = |V_b| = 5$
— $E_a = \{(v_1, v_2), (v_1, v_5), (v_2, v_2), (v_2, v_3), (v_3, v_1), (v_3, v_4), (v_3, v_5), (v_4, v_4), (v_5, v_2), (v_5, v_3), (v_5, v_4)\}$
— $E_b = \{(v_1, v_2), (v_1, v_3), (v_1, v_5), (v_2, v_1), (v_2, v_2), (v_2, v_3), (v_2, v_5), (v_3, v_1), (v_3, v_2), (v_3, v_4), (v_3, v_5), (v_4, v_3), (v_4, v_4), (v_4, v_5), (v_5, v_1), (v_5, v_2), (v_5, v_3), (v_5, v_4)\}$

4.1.3 Définitions générales

Définition 3.

- Le graphe $G = (V, E)$ est dit orienté symétrique si la présence de tout arc implique la présence de l'arc opposé.
- Pour un arc (v_i, v_j) de E, le sommet v_i est son extrémité initiale, et le sommet v_j son extrémité finale.
- Une arête (ou un arc) (v_i, v_j) est multiple, si elle correspond à plusieurs arêtes (ou arcs) ayant v_i comme extrémité initiale et v_j comme extrémité finale, dans ce cas là, on parle d'un multigraphe.
- Le graphe $G = (V, E)$ est dit sans boucle si E ne contient pas d'arêtes (ou arcs) de la forme (v, v), c-à-d joignant un sommet à lui même.
- On dit qu'un graphe $G = (V, E)$ est pondéré si chaque arc de G est menu d'un nombre, appelé poids de cet arc. En général, le poids d'un arc représente un coût donné.
- On dit qu'un sommet v_j est voisin au sommet v_i si $(v_i, v_j) \in E$. L'ensemble des voisins de v_i se note $N_G(v_i)$.
- Deux sommets v_i et v_j sont adjacents l'un à l'autre si l'un est voisin à l'autre. Dans ce cas, on dit que l'arête (v_i, v_j) est incidente à v_i et à v_j.
- On appelle degré d'un sommet v_i et on note $d_G(v_i)$, que l'on simplifie en $d(v_i)$ le nombre d'arêtes incidentes à v_i qui est aussi le nombre de voisins de v_i, c'est-à-dire $d(v_i) = |N_G(v_i)|$.
- Le maximum (resp. minimum) des degrés des sommets est appelé degré maximum (resp. minimum) d'un graphe, noté Δ (resp. δ). Si $\Delta = \delta$, le graphe est dit Δ-régulier.

4.1.4 Chemin et chaîne

Définition 4.

Dans un graphe orienté, un chemin allant d'un sommet x vers un sommet y, que l'on note souvent par $P_n = (x, y)$, est une suite finie de n sommets $(v_1, v_2, ..., v_n)$ tels que $x = v_1$, $y = v_n$ et pour tout i dans $\{1, 2, ..., n-1\}$, v_i et v_{i+1} sont adjacents.

La longueur du chemin P_n, notée $\ell(P_n)$, est le nombre d'arcs de la suite $(v_1, v_2, ..., v_n)$, c-à-d $\ell(P_n) = n - 1$.

Un chemin qui ne rencontre pas deux fois le même sommet est dit élémentaire. Un chemin est dit simple si un arc a est présent au plus une fois. Le poids d'un chemin est la somme des poids des arcs qui le composent. Dans le cas d'un graphe non orienté, on parle d'une chaîne.

Remarque 1.

On parlera le plus souvent d'un chemin, sans spécifier qu'il est une chaîne, simple ou élémentaire, la distinction dépendant de la nature du graphe considéré.

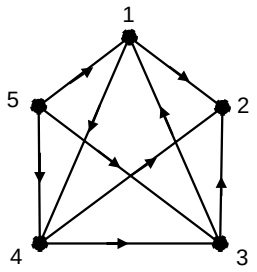
4.1.5 Distance et diamètre

Définition 5.

La distance entre deux sommets u et v d'un graphe G orienté (resp. non orienté), notée $d_G(u, v)$ ou simplement $d(u, v)$ est la longueur du plus court chemin (resp. chaîne) entre u et v.

On appelle diamètre d'un graphe $G = (V, E)$ (orienté ou non), noté $D(G)$, le maximum des distances entre toutes les paires de sommets de G : $D(G) = \max_{u,v \in V} d(u, v)$

Exemple : Les distances entre l'ensemble des sommets du graphe suivant



Sont

$$\begin{array}{llll} d(1,2)=1 & d(1,3)=2 & d(1,4)=1 & d(1,5)=\infty \\ d(2,1)=\infty & d(2,3)=\infty & d(2,4)=\infty & d(2,5)=\infty \\ d(3,1)=1 & d(3,2)=1 & d(3,4)=2 & d(3,5)=\infty \\ d(4,1)=2 & d(4,2)=1 & d(4,3)=1 & d(4,5)=\infty \\ d(5,1)=1 & d(5,2)=2 & d(5,3)=1 & d(5,4)=1 \end{array}$$

Il est facile de remarquer que le diamètre de ce graphe est 2.

4.2 Représentation algorithmique

En algorithmique, les graphes peuvent êtres considérés comme une structure de données. C'est pourquoi, il est fondamental de s'intéresser à la manière de les représenter en vue de leurs manipulations algorithmiques. Plusieurs modes de représentation peuvent être envisagés selon la nature des traitements que l'on souhaite appliquer au graphe considéré.

1. Représentation par matrice d'adjacences.
2. Représentation par tableau de listes d'adjacences.

4.2.1 Représentation par matrice d'adjacences :

La matrice d'adjacence d'un graphe simple $G = (V, E)$ d'ordre n est la matrice

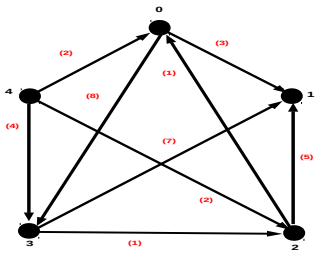
booléenne $M = (m_{ij})_{1 \leq i,j \leq n}$ de dimension $n \times n$ telle que

$$m_{ij} = \begin{cases} 1, & \text{si } (i,j) \in E \text{ (c-à-d } (i,j) \text{ est un arc)}\,; \\ 0, & \text{sinon.} \end{cases}$$

Si le graphe n'est pas pondéré ou

$$m_{ij} = \begin{cases} p_{ij}, & \text{si } (i,j) \in E \text{ et } p_{ij} \text{ est le poids de } (i,j)\,; \\ 0, & \text{sinon.} \end{cases}$$

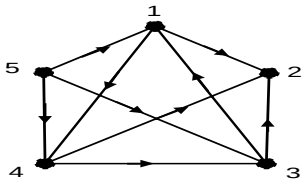
Le graphe simple non orienté suivant



a pour matrice d'adjacences

$$\begin{pmatrix} 0 & 3 & 0 & 8 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 5 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 2 & 0 & 2 & 4 & 0 \end{pmatrix}$$

Le graphe orienté suivant



a pour matrice d'adjacences

$$\begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Remarque 2.

- Un graphe orienté quelconque a une matrice d'adjacence quelconque, alors qu'un graphe non orienté possède une matrice d'adjacence symétrique.
- L'absence de boucle se traduit par une diagonale nulle.
- Un intérêt de cette représentation est que la détermination de chemins dans un graphe G revient au calcul des puissances successives de la matrice M. En effet, Soit $M^p = (m_{ij}^p)$ la puissance p-ième de la matrice M.
  - Si M est définie comme matrice booléenne, on a

$$m_{ij}^p = \begin{cases} 1, & \text{s'il existe un chemin de longueur } p \text{ de } i \text{ à } j\,; \\ 0, & \text{sinon.} \end{cases}$$

  - Si M est définie comme une matrice carrée dont les coefficients sont 0 et 1, on a le coefficient m_{ij}^p est égal au nombre de chemins de longueur p de G dont l'origine est le sommet i et dont l'extrémité est le sommet j.
- la somme des éléments de la i-ème ligne de M est égale au degré sortant $d_s(i)$ du sommet i de G.
- la somme des des éléments de la j-ème colonne de M est égale au degré entrant $d_e(j)$ du sommet j de G.

Codage en C

Définition d'un graphe : Cette structure permet de définir un graphe par sa matrice d'adjacences.

```
typedef struct
 {
  int nbrS; // <-- nombre de sommets
  int **m;  // <-- matrice d'adjacences
 }graphe;
```

Définition d'un arc : Cette structure permet de définir un arc d'un graphe.

```
typedef struct
 {
  int s,d,p;
 }arc;
```

Fonction d'ajout d'un arc : Cette fonction permet d'ajouter un arc à un graphe.

```
graphe ajoutArc(arc e, graphe g)
{
 g.m[e.s][e.d]=e.p;
 return g;
}
```

Fonction d'initialiser un graphe :

```
graphe intialiseGraphe(graphe g,int n)
{
 int i,j;
 g.nbrS=n;
 g.m=allocMat(n);
 for(i=0;i<g.nbrS;i++)
     for(j=0;j<g.nbrS;j++)
         g.m[i][j]=0;
 return g;
}
```

Fonction de construction d'un graphe à partir d'un fichier :

```
graphe constGrapheFile(FILE *f)
{
 graphe g;
 arc e;
 int n;
 fscanf(f,"%d",&n);
 g=intialiseGraphe(g,n);
 while(fscanf(f,"%d%d%d",&e.s,&e.p,&e.d)==3)
   g=ajoutArc(e,g);
return g;
}
```

4.2.2 Application : distance et plus court chemin

Soit $G = (V, E)$ un graphe orienté (ou non) d'ordre n avec $V = \{0, 1, \cdots, n-1\}$ tel que chaque arc ait un poids. L'objectif ici est de présenté un algorithme qui calcule les distances entre les sommets de G, ainsi que des plus courts chemins.

4.2.2.1 Algorithme de Floyd-Warshall

Notons $D = (d_{ij})$ la matrice des distances dans G d'ordre n telle que :

$$d_{ij} = \begin{cases} \text{le poids de l'arc (i,j)}, & \text{si } (i,j) \in E\,; \\ 0, & \text{si } i = j\,; \\ \infty, & \text{sinon}\,; \end{cases}$$

Pour $0 \leq k \leq n-1$, notons $D^{(k)} = (d_{ij}^{(k)})$ la matrice d'ordre n dont le terme $d_{ij}^{(k)}$ représente le poids minimal d'un chemin d'origine i et d'extrémité j, et vérifie la condition que tous les sommets intermédiaires appartiennent au sous-ensemble $\{0, 1, \cdots, k\}$.

L'état initial est la matrice $D = (d_{ij})$ avec d_{ij} est le poids de l'arc entre i et j où il n'y pas de sommet intermédiaire.

Principe de l'algorithme L'algorithme repose sur la remarque suivante :

— Si $(i, \cdots, j)$ est un plus court chemin de i à j et x un sommet intermédiaire, alors $(i, \cdots, x)$ est un plus court chemin de i à x, et $(x, \cdots, j)$ un plus court chemin de x à j.

— Deux situations peuvent se produire suivant que le plus court chemin de i à j à sommets intermédiaires dans $\{0, 1, \cdots, k\}$ emprunte le sommet k ou non.

1. Si oui, ce chemin est formé d'un sous-chemin entre i et k, suivi d'un sous-chemin entre k et j, chacun ne pouvant utiliser comme sommets intermédiaires que des sommets de $\{0, 1, \cdots, k-1\}$ et devant être de longueur minimale. On doit donc avoir
$$d_{ij}^{(k)} = d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$$
2. Sinon, on doit évidemment avoir
$$d_{ij}^{(k)} = d_{ij}^{(k-1)}$$

Ainsi, l'algorithme de Floyd-Warshall montre qu'il suffit de calculer la suite de matrices définies par :
$$d_{ij}^{(k)} = min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) \tag{4.1}$$
et la matrice $D^{(n-1)}$, donnant l'ensemble des valeurs des plus courts chemins dans le graphe G, pourra donc déterminée en n étapes de récurrence à partir de (4.1).

Fonction qui donne la matrice $D^{(0)}$:

```
int ** matDist(graphe g)
{
 int i,j,k,**m,n=g.nbrS;
 m=allocMat(n);
 for(i=0;i<n;i++)
  for(j=0;j<n;j++)
   {
    if(i==j)
      m[i][j]=0;
```

```
    else if(g.m[i][j]!=0)
            m[i][j]=g.m[i][j];
          else
            m[i][j]=infini;
   }
 return m;
}
```

Fonction qui donne la matrice des prédécesseur d'un sommet :

```
int ** matPrc(graphe g)
{
 int i,j,k,**m,n=g.nbrS;
 m=allocMat(n);
 for(i=0;i<n;i++)
  for(j=0;j<n;j++)
   {
    if(i==j || g.m[i][j]!=0)
      m[i][j]=i;
    else
      m[i][j]=-1;
   }
 return m;
}
```

Fonction qui donne le plus court chemin :

```
int **Floyd(graphe g,int ** prc)
{int i,j,k,**m,n=g.nbrS;
 m=matDist(g);
 for(k=0;k<n;k++)
  for(i=0;i<n;i++)
   for(j=0;j<n;j++)
   {
    if(m[i][j]>m[i][k]+m[k][j])
      prc[i][j]=prc[k][j];
    m[i][j]=min(m[i][j],m[i][k]+m[k][j]);
   }
 return m;
}
```

Fonction qui donne itinéraire du plus court chemin d'un sommet à un autre :

```
void chemin(int s, int d,graphe g)
{
 int ** prc,**dist,*p,t,i=0,j;
prc=matPrc(g);
dist=Floyd(g,prc);
p=malloc(g.nbrS*sizeof(int));

if(dist[s][d]!=infini)
 {
  p[i++]=d;
  t=d;
  while(s!=t)
  {
   t=prc[s][t];
   p[i++]=t;
  }
  printf("Le plus court chemin entre %d et %d est \n",s,d);
  for(j=i-1;j>=1;j--)
   printf("%d-->",p[j]);
  printf("%d",p[j]);
 printf(" de poids %d\n",dist[s][d]);
}
else
 printf("Pas de  chemin entre %d et %d !!!\n",s,d);
}
```

Remarque 3.

Le temps d'exécution de l'algorithme de Floyd est déterminé par les trois boucles `pour` imbriquées.

L'instruction dans la 3 ème boucle prend un temps $\mathcal{O}(1)$. Le temps d'exécution total de `FLOYD` est donc $\Theta(n^3)$.

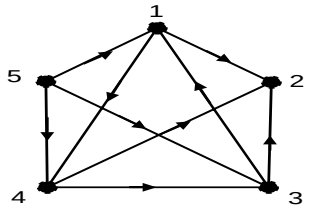
4.2.3 Représentation par tableau de listes d'adjacences :

Les graphes complets sont les seuls classes de graphes où tous les sommets sont adjacents deux à deux. Pour le reste des graphes il y a toujours des couples de sommets qui ne sont pas adjacents, ainsi leurs matrices d'adjacences peut contenir plusieurs zéro ce qui entraîne une utilisation de la mémoire sans utilité. Pour représenter un graphe en optimisant la mémoire, on propose une altérative des matrices d'adjacences, les tableaux de listes d'adjacences qui permettent de stocker seulement les arcs existants dans un graphe.

Soit un graphe $G = (V, E)$ de n sommets. le tableau de listes d'adjacences de G est un tableau T de n listes, une pour chaque sommet de V. Un tel tableau doit vérifier pour tout couple de sommets (i, j) :

$$j \in T[i] \Leftrightarrow (i, j) \in E$$

Exemple : Le graphe suivant



a pour tableau de listes d'adjacences :

$$T[1] = (2, 4),\ T[2] = (),\ T[3] = (2, 1),\ T[4] = (2, 3),\ T[5] = (1, 3, 4)$$

Remarque 4.

- Cette représentation est utile pour obtenir tous les successeurs d'un sommet i.
- Elle permet d'y accéder en un nombre d'opérations égal au nombre d'éléments de cet ensemble et non pas, comme c'est le cas dans la matrice d'adjacence, au nombre total de sommets.
- Exemple : si dans un graphe de 1000 sommets chaque sommet n'a que 5 successeurs l'obtention de tous les successeurs de i se fait en consultant 4 ou 5 valeurs au lieu des 1000 tests à effectuer dans le cas des matrices d'adjacences.
- Il y a différentes manières de coder cette représentation, par exemple on peut utiliser une structure de liste chaînée ou un un tableau à double indice ($TabSucc[i][j]$ =le j-ème successeur de i).

Codage en C

Définition d'un sommet : Cette structure permet de définir un sommet qui contient le poids de l'arc (dans le cas où il s'agit du sommet d'extrémité).

```
typedef struct {
  int indice, poid;
 }sommet;
```

Définition d'une liste de sommets : Cette structure permet de définir une liste de sommets (les voisins d'un sommet).

```
struct cellule{
  sommet s;
  struct cellule *suivant;
 };
typedef struct cellule cellule, *listAdj;
```

Définition d'un arc : Cette structure permet de définir un arc d'un graphe.

```
typedef struct{
  int s,d,p;
 }arc;
```

Définition d'un graphe : Cette structure permet de définir un graphe par son tableau de liste d'adjacences.

```
typedef struct
 {
  int nbrS;
  listAdj *tabAdj;
 }graphe;
```

Fonction d'ajout d'un arc : Cette fonction permet d'ajouter un arc à un graphe.

```
graphe ajoutArc(arc e, graphe g)
{
 sommet S;
 S.indice=e.d;
 S.poid=e.p;
 g.tabAdj[e.s]=ajoutSommetListe(g.tabAdj[e.s],S);
 return g;
}
```

Fonction d'initialiser un graphe :

```
graphe intialiseGraphe(graphe g,int n)
{
 int i,j;
 g.nbrS=n;
 g.tabAdj=malloc(g.nbrS*sizeof(cellule));
 for(i=0;i<g.nbrS;i++)
    g.tabAdj[i]=NULL;
 return g;
}
```

Fonction de construction d'un graphe à partir d'un fichier :

```
graphe constGrapheFile(FILE *f)
{
 graphe g;
 arc e;
 int n;
 fscanf(f,"%d",&n);
 g=intialiseGraphe(g,n);
 while(fscanf(f,"%d%d%d",&e.s,&e.p,&e.d)==3)
   g=ajoutArc(e,g);
return g;
}
```

4.3 Parcours en largeur, Parcours en profondeur

4.3.1 Présentation

Un parcours de graphe est une méthode systématique d'exploration des sommets et des arêtes d'un graphe. Le principe est toujours le même : on part d'un sommet et on effectue une promenade en choisissant les arêtes. On distingue particulièrement deux méthodes de parcours :

1. Le parcours en largeur.
2. Le parcours en profondeur.

Ces parcours permettent de Reconnaître certaines propriétés importantes, par exemple : connexité, accessibilité des sommets, etc. et notons qu' un grand nombre de problèmes sur les graphes admet pour solutions des algorithmes utilisant ces deux parcours, par exemple le problème du plus court chemin, calcul des distances, etc.

4.3.2 Parcours en largeur

4.3.2.1 Principe

Parcourir en largeur un arbre a pour effet de parcourir la racine, puis les sommets de hauteur 1, puis ceux de hauteur 2 ainsi de suite. Le parcours en largeur d'un graphe

$G = (V, E)$ à partir d'un sommet s est similaire. On parcourt s, puis ses voisins, puis les voisins de ses voisins non déjà visités et ainsi de suite. L'objectif est de parcourir un graphe à partir d'un sommet origine s, de calculer les distances des sommets parcourus à s et de construire une arborescence en largeur de racine s.

Remarque 5.

On découvre les sommets situés à la distance k avant les sommets situés à la distance $k+1$. Les sommets non atteignables à partir de l'origine ne sont jamais découverts.

4.3.2.2 L'algorithme et le code en C

L'algorithme du parcours en largeur suppose que le graphe d'entrée $G = (V, E)$ est représenté par un tableau de listes d'adjacences. L'idée pour mémoriser les sommets visités est d'attribuer une couleur parmi trois (blanche : sommet non visité, grise sommets en cours de visite et noire sommet déjà visité) à chaque sommet $x \in V$ et qu'est stockée dans l'élément du tableau `couleur[x]`.

Le prédécesseur (ou père) d'un sommet `x` est stocké dans l'élément du tableau `pred[x]`. Si `x` n'a pas de prédécesseur (par exemple, si `x=s` ou si `x` n'a pas été découvert), alors `pred[x]=NIL`.

La distance calculée par l'algorithme entre `s` et le sommet `x` est stockée dans l'élément du tableau `dist[x]`.

Signalons que l'algorithme utilise également une file `f` pour gérer l'ensemble des sommets gris.

Parcours en largeur : Dans le programme suivant on présente une définition d'une file d'entiers. La fonction `enfiler()` permet d'enfiler un entier dans une file, la fonction `defiler()` permet de défiler et récupérer la valeur défilée dans une file. La fonction `parcourLargeurGraph()` permet de parcourir un graphe à partir d'un sommet d'origine et d'avoir les distances des sommets à ce sommet et les sommets qui composent le chemin (stockés dans le tableau `pred[]`) parcouru entre le sommet d'origine et un sommet cible.

```
struct file{
 int valeur;
 struct file * suivant;
};
typedef struct file file, * File;

File enfiler(int s, File f)
{File fn;
 fn=malloc(sizeof(file));
 fn->valeur=s;
 fn->suivant=f;
 return fn;
}

int defiler(File *f)
{int vdef;
 File pfile1, pfile2;
 if((*f)->suivant==NULL)
  {
   vdef=(*f)->valeur;
   free(*f);
   *f=NULL;
   return vdef;
  }
 pfile1=*f;
 while(pfile1->suivant->suivant!=NULL)
    pfile1=pfile1->suivant;
 pfile2=pfile1->suivant;
 pfile1->suivant=NULL;
 vdef=pfile2->valeur;
 free(pfile2);
 pfile2=NULL;
 return vdef;
}

void parcourLargeurGraph(graphe g,int s,int * dist, int *pred)
{int i,k,*v;
 char  *couleur;
 File f=NULL;
 int x,y;
```

```
couleur=malloc(g.nbrS*sizeof(char));
for(i=0;i<g.nbrS;i++)
  if(i==s)
    dist[i]=0;
  else
    dist[i]=infini;
for(i=0;i<g.nbrS;i++)
    pred[i]=NIL;
for(i=0;i<g.nbrS;i++)
  if(i!=s)
    couleur[i]='B';
  else
    couleur[i]='G';
f=enfiler(s,f);
while(f!=NULL)
 {
  x=defiler(&f);
  v=voisinSommet(x,g,&k);
  if(k>1)
    printf("- le sommet visite est %d, ses successeurs sont : ",x);
 else if(k==1)
     printf("- le sommet visite est %d, son successeur est : ",x);
   else if(k==0)
    printf("- le sommet visite est %d, ne possede pas de successeur",x);
  for(i=0;i<k;i++)
   {
    printf("%d ",v[i]);
    if(couleur[v[i]]=='B')
     {
      couleur[v[i]]='G';
      dist[v[i]]=dist[x]+1;
      pred[v[i]]=x;
      f=enfiler(v[i],f);
     }
   }
  couleur[x]=='N';
  printf("\n");
 }
}
```

Remarque 6.

Dans la fonction `parcourLargeurGraph()` en visitant la boucle `while(f)` (la file contient des sommets non visités), le test de la couleur garantit que chaque sommet est enfilé au plus une fois, de même chaque sommet est défilé au plus une fois.

Puisque les opérations d'enfilement et de défilement sont en $\mathcal{O}(1)$, le temps total des opérations de file est $\mathcal{O}(|V|)$. Comme la liste d'adjacences de chaque sommet n'est balayée qu'au moment où le sommet est défilé, la liste d'adjacences de chaque sommet est parcourue au plus une fois. La somme des longueurs de toutes les listes d'adjacences étant $\Theta(|E|)$, le temps total consacré au balayage des listes d'adjacences est $\mathcal{O}(|E|)$.

Le coût d'initialisation est $\mathcal{O}(|V|)$.Ainsi Le temps d'exécution total de `ParcoursLargeur` est donc $\mathcal{O}(|V| + |E|)$.

Le parcours en largeur s'exécute en un temps qui est linéaire par rapport à la taille de la représentation par listes d'adjacences de G.

4.3.2.3 Applications

L'algorithme du parcours en largeur permet de trouver la distance entre une origine $s \in V$ donnée et chaque sommet accessible depuis s dans un graphe $G = (V, E)$. Autrement, cet algorithme permet de déterminer le plus court chemin entre un sommet $s \in V$ donné et chaque sommet accessible depuis s dans un graphe $G = (V, E)$. Cet algorithme permet également de construire une arborescence pendant qu'il parcourt le graphe. L'arborescence est représentée par le tableau `pred[]` de chaque sommet. Il permet aussi de construire un graphe partiel d'un graphe donné et qui contient tout les sommets accessible à partir d'un sommet origine.

4.3.3 Parcours en profondeur

4.3.3.1 Principe

Contrairement à l'algorithme du parcours en largeur, l'algorithme du parcours en profondeur permet de descendre plus profondément dans le graphe chaque fois que c'est possible. Les arcs sont explorés à partir du sommet s découvert le plus récemment et dont on n'a pas encore exploré tous les arcs incidents. Lorsque tous les arcs de s ont été explorés, l'algorithme revient en arrière pour explorer les arcs qui partent du sommet à partir duquel s a été découvert.

Ce processus se répète jusqu'à ce que tous les sommets accessibles à partir du sommet origine initial aient été découverts. S'il reste des sommets non découverts, on en choisit un qui servira de nouvelle origine, et le parcours reprend à partir de cette origine. Le processus complet est répété jusqu'à ce que tous les sommets aient été découverts.

Remarque 7.

Contrairement au parcours en largeur, pour lequel le sous-graphe de liaison forme une arborescence, le sous-graphe de liaison obtenu par un parcours en profondeur peut être composé de plusieurs arborescences, car le parcours peut être répété à partir de plusieurs origines.

4.3.3.2 L'algorithme et le code en C

L'algorithme du parcours en profondeur permet de créer une forêt de parcours en profondeur et *date* chaque sommet. En effet, chaque sommet x porte deux dates : la première stockée dans le tableau `debut[]`, marque le moment où x a été découvert pour la première fois (coloré en gris), et la seconde, dans le tableau `fin[]`, marque le moment ou le parcours a fini d'examiner la liste d'adjacences de x (coloré en noir).

4.3.3.3 L'algorithme et le code en C

L'algorithme du parcours en profondeur suppose que le graphe d'entrée $G = (V, E)$ est représenté par des listes d'adjacences. Cet algorithme enregistre le moment où elle découvre le sommet x dans la variable `debut[x]`, et le moment où elle termine le traitement de sommet x dans la variable `fin[x]`. Ces dates sont des entiers compris entre 1 et $2|V|$, puisque découverte et fin de traitement se produisent une fois et une seule pour chacun des $|V|$ sommets. Pour tout sommet x, $debut[x] < fin[x]$. Cet algorithme utilise également des couleurs (Blanche, grise, noire) pour mémoriser les sommets visités. Ainsi, le sommet x est `BLANC` avant l'instant `debut[x]`, `GRIS` entre `debut[x]` et `fin[x]`, et `NOIR` après.

Finalement, notons que le graphe d'entrée peut être orienté ou non.

Parcours en profondeur : Dans le programme suivant on présente deux fonctions, `explorer()` et `ParcoursProfondeur()` qui coopèrent pour réaliser le parcours en profondeur d'un graphe.

```
void explorer(graphe g,int x,char *coul,int *date,int *pred,int *debut,int *fin)
 {int i,k=0,*v;
  listAdj l=g.tabAdj[x];
  coul[x] ='G';
  (*date)++;
  debut[x]=*date;

while(l!=NULL)
  {
    if(coul[l->s.indice]=='B')
      {
       pred[l->s.indice]=x;
       explorer(g,l->s.indice,coul,date,pred,debut,fin);
      }
   l=l->suivant;
  }
  coul[x]='N';
  (*date)++;
  fin[x]=*date;
 }
```

```
void ParcoursProfondeur(graphe g,int *debut,int *fin,int *pred)
 {
  int i,date;
  char *coul;
  coul=malloc(g.nbrS*sizeof(char));
  for(i=0;i<g.nbrS;i++)
    {
      pred[i]=NIL;
      coul[i]='B';
    }
  date=0;
 for(i=0;i<g.nbrS;i++)
   if(coul[i]=='B')
     explorer(g,i,coul,&date,pred,debut,fin);
 }
```

Remarque 8.

Les deux boucles de la fonction `ParcoursProfondeur()` demandent un temps $\Theta(|V|)$ et la fonction `explorer()` est appelée exactement une fois pour chaque sommet $x \in V$, puisqu'elle n'est invoquée que sur les sommets blancs, et qu'elle commence par les colorer en gris. Pendant l'exécution de `explorer()`, la boucle `while(1)` est exécutée p fois où $p(x)$ est le nombre de voisins du sommet x. Comme

$$\sum_{x \in V} |p(x)| = \Theta(E)$$

le coût total d'exécution de `explorer()` est $\Theta(E)$.

Le temps d'exécution total de `ParcoursProfondeur()` est donc $\mathcal{O}(|V| + |E|)$.

Le parcours en profondeur s'exécute en un temps qui est linéaire par rapport à la taille de la représentation par listes d'adjacences de G.

4.3.3.4 Applications

L'algorithme du parcours en profondeur permet de reconnaître si un graphe est acyclique (sans cycle). De même il permet de réaliser le tri topologique et de déterminer les Composantes fortement connexes d'un graphe orienté.

4.4 Plus courts chemins : Algorithme de Dijkstra

4.4.1 Motivation et principe

Les problèmes de cheminement dans les graphes (en particulier la recherche d'un plus court chemin) comptent parmi les problèmes les plus anciens de la théorie des graphes et les plus importants par leurs applications. Ce problème permet la modélisation de plusieurs problèmes, On peut citer entre autres :

- les problèmes d'optimisation de réseaux (routiers, télécommunications),
- certaines méthodes de traitement numérique du signal, de codage et de décodage de l'information,
- certains problèmes d'investissement et de gestion de stocks,
- etc.

Soit $G = (V, E)$ un graphe muni d'une fonction poids ω qui associe à chaque arc $a = (i, j)$ un poids $\omega(a)$ ou ω_{ij}. Le problème du plus court chemin entre i et j est de trouver un chemin $P(i, j)$ de i à j tel que $\omega(P) = \sum_{(i,j)\in P} \omega_{ij}$ soit minimal.
$\omega(P)$ peut représenter soit le coût de transport, soit la dépense de construction, soit le temps nécessaire de parcours,...

4.4.1.1 L'algorithme et le code en C

Soit $G = (V, E)$ étant un graphe orienté d'ordre n avec $V = \{1, 2, \cdots, n\}$ et $\omega_i \geq 0$ la longueur minimum des chemins du sommet 1 au sommet i ; en particulier $\omega_i(1) = 0$. Notons par $\omega(x, y)$ = longueur (poids) de l'arc (x, y).

Le graphe G est représenté par des listes d'adjacences. F désigne un ensemble (liste)

de sommets dont les longueurs finales de plus court chemin à partir de l'origine s ont déjà été calculées.

Algorithme de Dijkstra : Dans le programme suivant on présente plusieurs fonctions qui permettent de réaliser l'algorithme de Dijkstra. En effet :

- `poidsArc(int s,int d,graphe g)` donne le poids de l'arc (s, d).
- `initialiseL(int n)` initialise la liste des sommets à traiter.
- `relacher(int s,int d,int *dis,int *pred,graphe g)` permet de redéfinir le poids d'un chemin entre les sommets (s, d) en choisissant celui de poids minimal.
- `extraireMin(liste l,int *dis,int *p)` permet de supprimer de la liste le sommet dont le poids est minimal.
- `Dijkstra(graphe g,int s,int *dis,int *pred)` permet à partir d'un sommets source s de trouver le plus court chemin avec le reste des sommets du graphe.

```
int poidsArc(int s,int d,graphe g)
{
 listAdj l;
 l=g.tabAdj[s];
 if(s==d)
   return 0;
 while(l!=NULL && l->s.indice!=d)
   l=l->suivant;
 if(l==NULL)
   return infini;
 return l->s.poid;
}

liste initialiseL(int n)
{
 liste l=NULL;
 int i=0;
 while(i<n)
  {
   l=ajoutDListe(i,l);
   i++;
  }
```

```
 return l;
}

void relacher(int s,int d,int *dis,int *pred,graphe g)
{
 if(dis[d]>dis[s]+poidsArc(s,d,g))
  {
    dis[d]=dis[s]+poidsArc(s,d,g);
    pred[d]=s;
  }
}

liste extraireMin(liste l,int *dis,int *p)
{
int m;
 liste lp=l;
 m=dis[lp->valeur];
 *p=lp->valeur;
 lp=lp->suivant;
 while(lp!=NULL)
  {
   if(m>dis[lp->valeur])
    {
     m=dis[lp->valeur];
     *p=lp->valeur;
    }
   lp=lp->suivant;
  }
 l=supprimeEltListe(*p,l);
 return l;
}

void Dijkstra(graphe g,int s,int *dis,int *pred)
{
 int x;
 liste F=initialiseL(g.nbrS);
 listAdj l;
 while(F!=NULL)
  {
   F=extraireMin(F,dis,&x);
   l=g.tabAdj[x];
```

```
    while(l!=NULL)
     {
      relacher(x,l->s.indice,dis,pred,g);
      l=l->suivant;
     }
   }
}
```

Chapitre 5

Exercices

Exercice 1 :

Écrire un programme complet en C permettant de réaliser quelques manipulations de base sur les arbres binaires notamment les tâches suivantes :

1. Définir une structure représentant un arbre binaire.
2. Construire un arbre binaire, un ABR et un AVL.
3. Afficher les éléments de chaque nœud d'un arbre binaire (en utilisant différentes méthodes).
4. Rechercher un élément dans un arbre binaire, ABR.
5. Supprimer un nœud dans un ABR et dans arbre quelconque. Vider un arbre binaire.

Exercice 2 :

1. Écrire une fonction qui teste si un arbre binaire est un arbre binaire ABR ?
2. Écrire une fonction qui permet de donner le nombre de nœuds par niveau.
3. Écrire une fonction itérative qui permet de donner le minimum/maximum d'un arbre binaire.
4. Écrire une fonction itérative qui permet de donner le minimum/maximum d'un ABR.

5. Écrire une fonction récursive qui permet de donner le minimum/maximum d'un arbre binaire.
6. Écrire une fonction récursive qui permet de donner le minimum/maximum d'un ABR.
7. Écrire une fonction récursive qui permet de calculer le nombre d'occurrences d'un entier dans un arbre binaire.
8. Écrire une fonction itérative qui permet de calculer le nombre d'occurrences d'un entier dans un arbre binaire.
9. Écrire une fonction récursive qui permet de calculer le nombre d'occurrences d'un entier dans un ABR.
10. Écrire une fonction itérative qui permet de calculer le nombre d'occurrences d'un entier dans un ABR.
11. Écrire une fonction qui permet de calculer le nombre d'occurrences d'un entier dans ABR à partir d'un niveau.

Exercice 3 :

Écrire des fonctions en C qui permettent de réaliser les tâches suivantes :

1. teste si un arbre binaire est un arbre binaire ABR ?
2. donne le nombre de nœuds et le nombre de feuilles par niveau,
3. calcule (méthode récursive) le nombre d'occurrences d'un entier dans un arbre binaire,
4. calcule (méthode itérative) le nombre d'occurrences d'un entier dans un arbre binaire,
5. calcule (méthode récursive) le nombre d'occurrences d'un entier dans un ABR,
6. calcule (méthode itérative) le nombre d'occurrences d'un entier dans un ABR,
7. calcule le nombre d'occurrences d'un entier dans ABR à partir d'un niveau.

Exercice 4 :

L'objectif de cet exercice est d'écrire un programme en C qui permet de manipuler un arbre binaire de recherche (ABR) de mots (chaîne de caractères sans espaces). Pour Ceci, définir la structure d'un arbre binaire de mots et écrire des fonctions qui permettent de

1. construire un ABR à partir d'un fichier,
2. d'afficher en ordre lexicographique les mots stockés dans ABR,
3. de donner le nombre de mots, commençant par un caractère donné, stockés dans un ABR,
4. d'afficher les mots qui sont stockés dans les feuilles d'un ABR,
5. d'afficher les mots d'un niveau donné d'un ABR,

Exercice 5 :

L'objectif de cet exercice est de présenter une méthode de tri de tableaux en utilisant la structure d'arbre binaire croissant. On dit qu'un arbre binaire `a` est un arbre croissant si, soit `a=V`, soit `a=N(fG,val,fD)` où `fG` et `fD` sont eux-même deux arbres croissants et `val` est inférieur ou égal à tous les éléments de `fG` et `fD`.

1. Écrire une fonction en C qui permet de tester si un arbre binaire est croissant.
2. Écrire une fonction en C qui permet de donner le minimum d'un arbre croissant.
3. L'opération de fusion de deux arbres croissants a_1 et a_2, noté $fusion(a_1, a_2)$, est définie récursivement de la manière suivante :

$$\begin{aligned}
fusion(a_1, V) &= a_1 \\
fusion(V, a_2) &= a_2 \\
fusion(N(fG_1, val_1, fD_1), N(fG_2, val_2, fD_2)) &= N(fusion(fD_1, N(fG_2, val_2, fD_2)), val_1, fG_1) \text{ si } val_1 \leq val_2 \\
fusion(N(fG_1, val_1, fD_1), N(fG_2, val_2, fD_2)) &= N(fusion(fD_2, N(fG_1, val_1, fD_1)), val_2, fG_2) \text{ sinon}
\end{aligned}$$

   Écrire une fonction en C qui permet de réaliser l'opération de fusion de deux arbres croissants.

4. Écrire une fonction en C qui permet l'ajout d'une valeur dans un arbre croissant. L'arbre obtenu doit être aussi un arbre croissant.

5. Écrire une fonction en C qui permet l'ajout d'une valeur dans un arbre croissant. L'arbre obtenu doit être aussi un arbre croissant.
6. Écrire une seule fonction en C qui permet de supprimer le plus petit élément d'un arbre croissant.
7. Écrire une fonction en C qui permet de copier les éléments d'un tableau dans un arbre croissant.
8. Écrire une fonction en C qui permet de trier un tableau d'entiers en utilisant un arbre croissant.

Exercice 6 : Tri par tas

Le tri par tas est une méthode rapide de tri de tableau. Le principe de ce tri repose sur la structure d'un tas. En effet, un tas est arbre binaire équilibré ou complet (toutes les feuilles sont à une profondeur égale à la hauteur ou à la hauteur -1 de l'arbre) et dont la valeur de chaque nœud est supérieure à celles des ses deux fils. L'idée de la méthode consiste à simuler ou à représenter un tableau `tab` par un tas où la valeur de la racine correspond à `tab[0]`, son fils gauche par `tab[1]` et le fis droit par `tab[0]` et ainsi de suite. La construction d'un tas permet d'avoir le maximum du tableau dans la racine ainsi en échangeant cette valeur avec celle la feuille de la branche de l'extrême droite on peut avoir le la valeur maximale dans la dernière case du tableau. Ainsi d'une manière récursive, on tri le tableau.
Ecrire un programme en C qui permet de trier un tableau avec la méthode de tri par tas.

Exercice 7 :

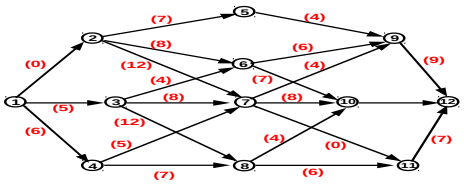
Écrire un programme complet en C permettant de réaliser quelques manipulations de base sur les graphes notamment les tâches suivantes :

1. Définir un graphe. Un graphe est représenté par sa matrice d'adjacences.
2. Construire un graphe à partir d'un fichier.
3. Tester si un graphe et orienté ou non.

4. Donner le degré d'un sommet donné. Noter que dans le cas d'un graphe orienté, le degré d'un sommet égal à la sommet des degrés entrants et degrés sortants du même sommet.
5. Donner le degré maximal et minimal d'un graphe.
6. Donner le plus court chemin dans graphe (Algorithme de Floyd-Warsall).
7. Donner la distance entre deux sommets dans un graphe.
8. Donner le diamètre d'un graphe.

Exercice 8 :

La ville de Safi compte réaliser un projet d'assainissement entre les quartiers 1 et 12. Les coûts de construction sont indiqués sur la figure ci-dessous. Les quartiers 1 et 2, et 7 et 11 sont déjà reliées par des canaux d'assainissement et si on utilise ces canaux le coût de construction est négligeable. Écrire un programme qui vous permet de déterminer les quartiers intermédiaires pour que le coût de construction soit minimal.



Exercice 9 :

Le tableau ci-dessous indique les connexions aériennes possibles entre les villes et les temps de vol en heures, correspondance incluse(ligne = ville de départ, colonne = ville d'arrivée. Ecrire un programme qui permet de trouver le vol qui dure moins de temps entre Casa et New York.

	Hambourg	Amsterdam	Londres	Berlin	Edimbourg	Oslo	Stockholm	New York
Casa	7	3	4					
Hambourg				1			1	
Amsterdam	2		1			8		
Londres					2			
Berlin		2				3	2	
Edimbourg		3				7		6
Oslo								2
Stockholm						2		5

Exercice 10 :

Écrire un programme complet en C permettant de réaliser quelques manipulations de base sur les graphes notamment les tâches suivantes :

1. Définir un graphe. Un graphe est représenté par un tableau de listes d'adjacences.
2. Construire un graphe à partir d'un fichier.
3. Tester si un graphe et orienté ou non.
4. Donner le degré d'un sommet donné. Noter que dans le cas d'un graphe orienté, le degré d'un sommet égal à la sommet des degrés entrants et degrés sortants du même sommet.
5. Donner le degré maximal et minimal d'un graphe.
6. Implémenter les algorithmes de parcours de graphes (Largeur et profondeur).
7. Donner le plus court chemin dans graphe (Algorithme de Dijkstra).
8. Donner la distance entre deux sommets dans un graphe.
9. Donner le diamètre d'un graphe.

Exercice 11 :

Traiter les problèmes de l'exercice 2 et 3 par les graphes représentés par un tableau de listes d'adjacences.