Polycopiés de cours

Algorithmique et Structures de données

Licence 2 d'informatique

Dr Abdelkamel, Ben Ali

benali-abdelkamel@univ-eloued.dz

Département d'informatique
Université d'El Oued

2019-2020

Table des matières

Chapitre 1 : Rappels — Structures de données linéaire : listes chaînes, piles et files 1

1.1 Notion de pointeurs 1

1.1.1 Passage des paramètres par adresse 2

1.1.2 Gestion dynamique de la mémoire 2

1.2 Listes chaînées 3

1.2.1 Définition 3

1.2.2 Implémentation en C des listes chaînées 3

1.2.3 Listes chaînées particulières 7

1.3 Piles 8

1.3.1 Structure et opérations 8

1.3.2 Implémentation d'une pile par une liste simplement chaînée 8

1.3.3 Quelques applications 9

1.4 Files 9

1.4.1 Structure et opérations 9

1.4.2 Implémentation d'une file par une liste chaînée circulaire 10

1.4.3 Quelques applications 10

Chapitre 2 : Complexité des algorithmes 11

2.1 Introduction 11

2.2 Complexité algorithmique 11

2.3 Principe de calcul 12

2.3.1 Modèles de calcul 12

2.3.2 Codages d'instances 13

2.3.3 Coût des opérations 13

2.3.4 Evaluation des coûts 14

Evaluation des coûts en séquentiel 14

Evaluation des coûts en récursif 15

2.4 Mesures et notations 15

2.4.1 Notations de la complexité 15

2.4.2 Notations utilises: Notations de Landau 17

Chapitre 3 : Algorithmes de tri 21

3.1 Présentation 21

3.2 Tri bulle 22

3.3 Tri par sélection 23
3.4 Tri par insertion 24
3.5 Tri fusion 25
3.6 Tri rapide '*quicksort*' 27
3.7 Complément de cours: recherche par dichotomie 30
Chapitre 4 : Les arbres 32
4.1 Introduction 32
4.2 Définitions 32
4.2.1 Représentation chaînée d'un arbre n-aire 33
4.3 Arbre binaire 34
4.3.1 Définition 34
4.3.2 Représentation chaînée d'un arbre binaire 34
4.3.3 Opérations sur un arbre binaire 35
4.3.4 Parcours d'un arbre binaire 36
4.3.5 Arbres binaires particuliers 37
Chapitre 5 : Les graphes 41
Introduction 41
5.1 Définitions 41
5.2 Représentations informatiques d'un graphe 44
5.3 Parcours d'un graphe 45
5.3.1 Parcours générique 46
5.3.2 Parcours en largueur 47
5.3.3 Parcours en profondeur 48

Chapitre 1

Rappels — Structures de données linéaire : listes chaînes, piles et files

L'objectif de cette partie de cours est décrire des structures de données linéaires de base telles que les **listes chainées** en général et deux formes particulières : les **piles** et les **files**. Nous commençons par présenter la notion de pointeurs et par décrire les fonctions C permettant la **gestion dynamique** de la mémoire.

1.1 Notion de pointeurs

- Une variable dans un programme désigne un "tiroir" dans la mémoire de l'ordinateur, dans lequel on peut stocker une valeur d'un certain type de données.
- Un **pointeur** est aussi un tiroir, une variable, mais pour stocker une ***référence*** à un autre tiroir.
- Le numéro d'un tiroir est appelé l'adresse de la variable. Si la variable x correspond à tel tiroir, et ce tiroir est le tiroir no 1700, un pointeur p qui fait référence à x sera un tiroir qui contiendra la valeur 1700.

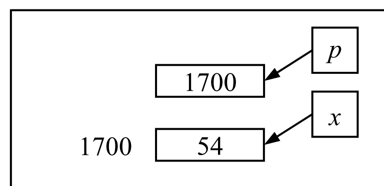


Fig. 1.1 : Une variable et un pointeur

En C, on utilise les notations suivantes :

- x pour désigner le contenu de x (54) ;
- p pour désigner le contenu de p (1700) ;
- $\&x$ pour désigner l'adresse du tiroir x (1700) ;
- $*\,p$ pour désigner le contenu du tiroir sur lequel pointe p (54).

1.1.1 Passage des paramètres par adresse

Dans certains cas, on souhaite que les paramètres soient effectivement modifiés lors d'un appel d'un sous-programme. Pour cela, la solution serait, plutôt que de communiquer la valeur, de communiquer l'adresse. Le passage par adresse permet de modifier les variables du (ou d'un sous) programme. C'est l'adresse de la variable, plutôt que sa valeur qui est transmise à un sous-programme.

- En C, pour le passage par adresse, il y a deux changements à opérer par rapport au passage par valeur :
  - Lors de l'appel, on met le signe `&` devant le paramètre pour désigner son adresse ;
  - Dans la fonction, on met le signe `*` devant le nom du paramètre pour désigner son contenu.
- En C, les tableaux sont toujours passés par adresse, et ceci sans notation particulière (pas de `&` et pas de `*`).

1.1.2 Gestion dynamique de la mémoire

Nous présentons les fonctions C qui permettent d'allouer dynamiquement la mémoire :

`sizeof`, `malloc`, `calloc`, `free`.

`sizeof (object)`

Donne la taille de l'objet.

`sizeof (nom de type)`

Donne la taille du type.

Exemple :

- `sizeof(char)` vaut 1 ;
- `sizeof(short)` vaut 2 (en principe) ;
- `sizeof(int)` : valeur dépendant de l'implantation (2 ou 4).

`malloc, calloc`

Ces fonctions allouent de la mémoire.

```
/* Tableau dynamique */
int *v1, *v2;
v1 = (int *) malloc (20 * sizeof (int));
v2 = (int *) calloc (20, sizeof (int));
```

De plus, `calloc` (mais pas `malloc` !) initialise la zone allouée avec des zéros. Ces fonctions sont définies dans `stdlib.h.`

`free(p)`

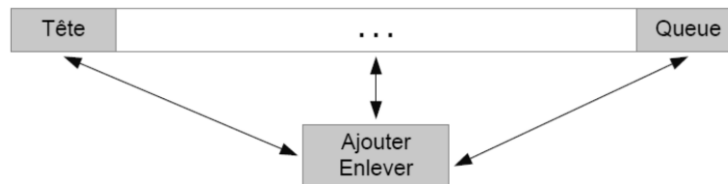
Libère la zone mémoire pointée par `p`, allouée par `malloc` ou `calloc`.

Chercher à libérer une zone mémoire qui n'a pas été allouée par `malloc` ou `calloc` provoque erreur (d'exécution).

1.2 Listes chaînées

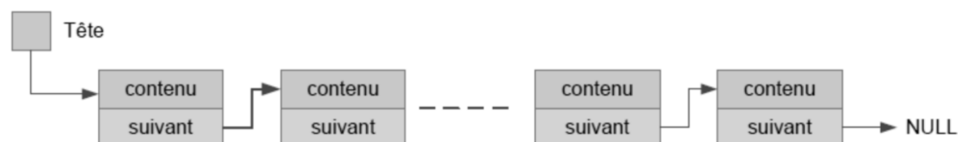
1.2.1 Définition

Les listes sont des structures de données qui permettent, au même titre que les tableaux, de stocker en mémoire des données de même type en respectant un certain ordre : on peut ajouter de nouvelles données, enlever des données, vider une liste, savoir si elle contient une valeur particulière, réarranger les données dans liste, etc.



1.2.1.1 *Liste chaînée*

- L'accès aux éléments dans une liste chaînée ne se fait pas par index (comme dans les tableaux) mais à l'aide d'un pointeur.
- Allocation dynamique de la mémoire : au moment de l'exécution, on alloue et libre les espaces mémoire.
- Une liste chaînée est une suite de couples formés d'un élément et de l'adresse (pointeur) vers l'élément suivant.
- Une liste chaînée est un pointeur vers le premier élément de la liste.
- Ce pointeur a la valeur nulle (NULL) si/quand la liste est vide.
- Les éléments sont contigus en ce qui concerne l'enchaînement, mais dans la mémoire chaque élément est stocké à un endroit quelconque.



1.2.2 Implémentation en C des listes chaînées

Nous commençons par les déclarations des types. Pour définir une cellule (élément) de la liste le type `struct` est utilisé. La cellule de la liste contient un champ contenu et un pointeur suivant.

```
typedef int Element;
struct Cellule
{
      Element contenu;
      struct Cellule *suivant;
};
typedef struct Cellule Cellule, *Liste;
```

Les opérations usuelles sur les listes sont :

- Créer une liste vide ou tester si une liste est vide ;
- Insérer un nouvel élément (au début de la liste, à la fin, derrière le k-ième élément, etc.) ;
- Rechercher un élément d'une valeur particulière ;
- Supprimer un élément d'une valeur particulière (ou le k-ième élément) ;
- Trier, afficher, concaténer, dupliquer …

Nous donnons ci-dessous les réalisations des opérations de base sur les listes chainées.

Création d'une liste vide

```
Liste FaireListeVide ()
{
	return NULL; /* NULL est défini dans <stdio.h> */
}
```

- Cette opération doit être effectuée avant toute autre opération sur la liste.
- Elle est utilisée pour initialiser le pointeur de type `Liste` avec le pointeur NULL.

Tester si une liste est vide

```
int EstListeVide (Liste a)
{
	if (a == NULL)
		return 1;
	else
		return 0;
}
```

- Cette fonction renvoie 1 si la liste est vide sinon elle renvoie 0.

Les opérations primitives sur les listes

Une opération pour renvoyer l'élément en tête de la liste.

```
Element tete (Liste a)
{
	return a->contenu;
}
```

Une autre opération pour renvoyer la liste queue de la liste.

```
Liste queue (Liste a)
{
	return a->suivant;
}
```

- Ces opérations primitives doivent être appliquées sur une liste non vide.

Insertion d'un nouvel élément dans la liste

Insertion au début de la liste

```
Liste Lajouter (Element x, Liste a)
{
	Liste b;
	b = (Liste) malloc (sizeof(Cellule));
	b->contenu = x;
	b->suivant = a;
	return b;
}
```

Initialisation d'une liste : Exemple

Initialisation d'une liste par les carrés des nombres $2..n$

```
Liste LCarres (int n)
{
	int i;
	Liste a;
	a = FaireListeVide ();
	for (i = n; i >= 2; i--)
		a = Lajouter (i*i, a);
	return a;
}
```

Recherche dans une liste (itérative)

Cette fonction renvoie 1 si la valeur x se figure dans la liste sinon elle renvoie 0.

```
int estDansI (Element x, Liste a)
{
	while (!EstListeVide (a)) /* liste non vide */
	{
		if (a->contenu == x)
			return 1;
		a = a-> suivant ;
	}
	return 0;
}
```

Recherche itérative (variante)

```
int estDansI (Element x, Liste a)
{
	for ( ; !EstListeVide (a); a = a->suivant)
		if (a-> contenu == x)
			return 1;
	return 0;
}
```

Recherche dans une liste (récursive)

```
int estDansR (Element x, Liste a)
{
	if (EstListeVide (a))
		return 0;
	else if (a->contenu == x)
		return 1;
	return estDansR (x, a->suivant);
}
```

Longueur (nombre d'éléments) d'une liste

```
/* De façon itérative */
int LlongueurIter (Liste a)
{
	int longueur = 0;
	while (!EstListeVide (a))
	{
		longueur ++;
		a = a->suivant ;
	}
	return longueur ;
}
/* De façon récursive */
int LlongueurRec (Liste a)
{
	if (EstListeVide (a)) return 0;
	else return 1 + LlongueurRec (a->suivant);
}
```

Suppression dans une liste (itérative)
de la première occurrence de x

```
Liste LsupprimerIter (Element x, Liste a)
{
	Liste b, c;
	if (!EstListeVide (a))
		if (a->contenu == x)
		{
			c = a;
			a = a->suivant;
			free (c);
		}
		else
		{
			b = a;
			while (!EstListeVide (b->suivant) &&
						b->suivant->contenu != x)
				b = b-> suivant;
			if (!EstListeVide (b->suivant))
			{
				c = b->suivant;
				b->suivant = b->suivant->suivant;
				free (c);
			}
		}
	return a;
}
```

Suppression dans une liste (récursive)
de la première occurrence de x

```
Liste LsupprimerRec (Element x, Liste a)
{
      Liste b;
      if (!EstListeVide (a))
      {
            if (a->contenu == x)
            {
                  b = a;
                  a = a->suivant;
                  free (b);
            }
            else a->suivant = LsupprimerRec (x, a-> suivant);
      }
      return a;
}
```

Destruction d'une liste

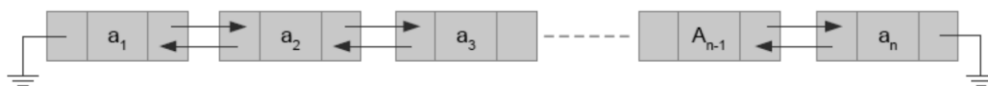
Cette opération est pour détruire la liste ; libérer les espaces mémoire occupés par la liste.

```
void detruire (Liste a)
{
      Cellule *courant;
      while (!EstListeVide(a))
      {
            courant = a;
            a = a->suivant;
            free (courant);
      }
}
```

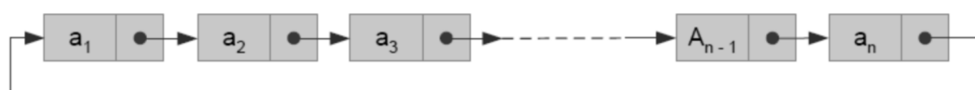
1.2.3 Listes chaînées particulières

Deux listes chaînées particulières sont :

1. ***Liste doublement chaînée*** : chaque élément de la liste contient une référence vers la liste suivante et une référence vers la liste précédente. La liste qui précède la tête de la liste est vide (NULL) et la liste qui suive le dernier élément est aussi vide (NULL).



2. ***Liste circulaire*** : le "suivant" du dernier élément est le premier élément.



1.3 Piles

1.3.1 Structure et opérations

Une pile (*stack*) est une structure de données linéaire, qui permet de stocker les données dans l'ordre **LIFO** (**L**ast **I**n **F**irst **O**ut – en français Dernier Entré Premier Sorti).

Une pile est une liste où les insertions et les suppressions des données se font toutes du même côté. La récupération des données sera faite dans l'ordre inverse de leur insertion.

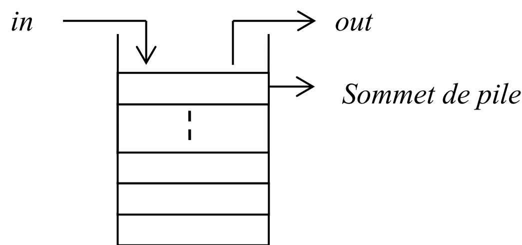


Fig. 1.2 : Structure pile

Cinq opérations sur une pile dont les éléments sont de type *élément,* sont :

PILEVIDE(p : pile) ;

Crée une pile vide p.

EST-PILEVIDE(p : pile) : booléen;

Teste si la pile p est vide.

SOMMET(p : pile) : élément ;

Renvoie l'élément au sommet de la pile p; p doit être non vide.

EMPILER(x : élément ; p : pile) ;

Insère la donnée dans x au sommet de la pile p.

DEPILER(p : pile) ;

Supprime l'élément au sommet de la pile p; Elle doit être appliquée sur une pile non vide.

Exercice interactif

Soit P une pile d'entiers, quel est son état après les opérations suivantes :
PILEVIDE(P) ; EMPILER(9, P) ; EMPILER(6, P) ; EMPILER(8, P) ; DEPILER(P) ;
EMPILER(SOMMET(P); P) ; EMPILER(9, P) ; DEPILER(P) ; DEPILER(P) ;

1.3.2 Implémentation d'une pile par une liste simplement chaînée

Une pile est réalisée par un pointeur vers un couple formé de l'élément au sommet de pile, et d'un pointeur vers le couple suivant (voir figure).

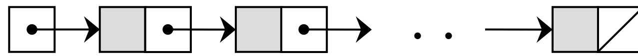


Fig. 1. 3 : Une pile représentée en utilisant une liste simplement chaînée

Les insertions (EMPILER) et suppressions (DEPILER) se font toutes au début de la liste.

1.3.3 Quelques applications

- Appels de fonctions, en particulier, les procédures récursives gèrent une pile de récursion ;
- Analyse syntaxique : une phase de la compilation qui nécessite une pile. Par exemple, la reconnaissance des expressions bien parenthésées ;
- Evaluation d'expressions arithmétiques ;

1.4 Files

1.4.1 Structure et opérations

La file (*queue*) une structure de données linéaire, qui permet de stocker les données dans l'ordre **FIFO** (**F**irst **I**n **F**irst **O**ut – en français Premier Entré Premier Sorti).

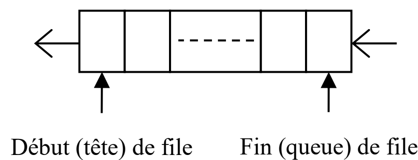


Fig. 1.4 : Structure file

Cinq opérations sur une file dont les éléments sont de type *élément*, sont:

FILEVIDE(f : file);

Crée une file vide f.

TETE(f : file) : élément ;

Renvoie l'élément en tête de la file f ; f doit être non vide.

ENFILER(x : élément ; f : file);

Insère la donnée dans x à la fin de la file f.

DEFILER(f : file);

Supprime l'élément en tête de la file f ; f doit être non vide.

EST-FILEVIDE(f : file) : booléen;

Teste si la file f est vide.

Les insertions (ENFILER) se font toutes d'un même côté et les suppressions (DEFILER) toutes de l'autre côté.

Exercice interactif

Soit F une file d'entiers, quel est son état après les opérations suivantes :
FILEVIDE(F) ; ENFILER(9, F) ; ENFILER(6, F) ; ENFILER(TETE(F), F) ; DEFILER(F) ; ENFILER(6, F) ; DEFILER(F) ;

1.4.2 Implémentation d'une file par une liste chaînée circulaire

Une file est repérée par un pointeur sur son dernier élément (si la file est vide, le pointeur vaut `NULL`). La tête de la file est l'élément suivant, et peut être facilement supprimée ; de même, l'insertion des nouvelles données en fin de la file se fait facilement.

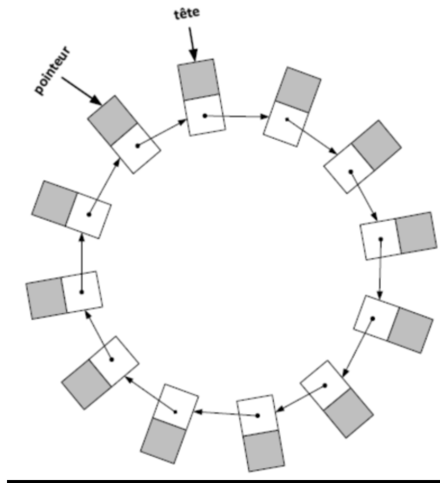


Fig. 1.5 : Une file représentée par une liste circulaire

1.4.3 Quelques applications

Les files permettent de gérer des processus en attente d'une ressource du système.

- Impression : maintenir une file de documents en attente d'impression ;
- Disque Driver : maintenir une file de requêtes d'input/output disque ;
- Ordonnanceur : maintenir une file de processus en attente d'un temps CPU.

Chapitre 2

Complexité des algorithmes

2.1 Introduction

L'objectif de cette partie de cours "Complexité des algorithmes" est :

- Présenter les notions et les principes liés à la complexité algorithmique.
- Fournir les outils mathématiques nécessaires à l'analyse des performances d'un algorithme.
- Donner une classification des problèmes de calcul du point de vue de leur complexité algorithmique.
- Montrer comment améliorer les performances des algorithmes faciles (TDs).

Motivation

- ***Définition***. Un **algorithme** est une procédure finie et mécanique de résolution d'un problème de calcul. *Exemples* : les algorithmes d'Euclide, l'algorithme de Dijkstra …
- Un algorithme doit se terminer sur toutes les *données* possibles du problème et doit retourner une **réponse**/***solution correcte*** dans chaque cas.
- Résoudre ***informatiquement*** un problème, c'est réaliser un algorithme sur un ordinateur en utilisant un langage de programmation.
  Mais, il existe bien souvent plusieurs algorithmes pour le même problème.
  Y a-t-il un intérêt à choisir un algorithme particulier ? Et si oui, quels sont les critères de choix ?
- En pratique, il n'est même pas suffisant de détenir un algorithme.
  Il existe des problèmes pour lesquels on a des algorithmes, mais qui restent comme informatiquement non résolus". C'est parce que le temps d'exécution est vite exorbitant.

2.2 Complexité algorithmique

Le mot complexité recouvre deux réalités : Complexité des algorithmes et Complexité des problèmes.

Complexité des algorithmes

C'est l'étude de l'efficacité comparée des algorithmes. On mesure le temps d'exécution CPU et aussi l'espace mémoire nécessaire à l'exécution d'un algorithme sur une machine :

- ***Complexité temporelle*** (plus intéressante)
- Complexité *spatiale*

Cela peut se faire de façon *expérimentale* ou *formelle*.

Complexité des problèmes

La complexité des algorithmes a abouti à une *classification* des problèmes de calcul en fonction des performances des meilleurs algorithmes connus pour leur résolution.

- La progression de la technologie ne change rien à cette classification ; elle a été établie indépendamment des caractéristiques techniques des ordinateurs.

Temps d'exécution d'un programme

On réalise un algorithme dans un langage de programmation haut niveau. Le temps d'exécution du programme dépend :

- des données du problème pour cette exécution (i.e., instance de problème)
- de la qualité du code engendre par le compilateur
- de la nature et de la rapidité des instructions offertes par l'ordinateur (technologie)
- de l'efficacité de l'algorithme conçu
- de l'encodage des données choisi
- de la qualité de la programmation !

A priori, on ne peut pas mesurer le temps de calcul sur toutes les entrées possibles d'un problème. Il faut trouver une autre méthode d'*évaluation*. L'idée est de s'affranchir des considérations subjectives (programmeur, caractéristiques techniques, …).

- On cherche une **grandeur** n pour "quantifier" les entrées.
- On évalue les **performances** (complexité) uniquement en fonction de n.

Définition. La fonction de complexité d'un algorithme fait correspondre pour une taille donnée le nombre d'instructions (opérations de base) qui lui est nécessaire pour résoudre une instance quelconque de cette taille.

$$T_{ALG}(n) = ?$$

2.3 Principe de calcul

2.3.1 Modèles de calcul

Les *machines de Turing* (TM) ou les *Random Access Machines* (RAM) (machines abstraites) servent d'"étalon" à la mesure des complexités en temps et en espace des fonctions dites calculables.

Pour TM

- Temps d'exécution : nombre de transitions nécessaires pour aboutir un état accepteur
- Espace mémoire requis : nombre de cases utilisées

Pour RAM

- Temps d'exécution : nombre d'opérations élémentaires : les mouvements en mémoire (READ, WRITE), les accès aux registres (LAOD, STORE), les opérations de base (ADD), les branchements (JUMP), etc.
  Programme = suite finie d'instructions codée sur la mémoire
- Espace mémoire : nombre de registres utilisées et leurs longueurs, indépendamment de leurs entrées-sorties.

Par la suite, on considère le calcul de la complexité en temps sur le modèle RAM. Notre approche pragmatique sert à comparer des algorithmes résolvant un même problème afin d'estimer rigoureusement lesquels sont les meilleurs.

2.3.2 Codages d'instances

La complexité temporelle dépend de la taille de l'entrée. En premier lieu, il faut déterminer la ***taille des données*** nécessaires à l'algorithme. On les appelle les *entrées* ou les *instances*. On veut considérer seulement la ***taille essentielle*** de l'entrée.

Exemples, selon que le problème est modélise par :

- des nombres : ces nombres
- des polynômes : le degré, le nombre de coefficients non nuls
- des matrices $n \times m$: $\max(n, m)$, $n + m, n \cdot m$
- des graphes : ordre, taille, produit des deux
- des arbres : comme les graphes et la hauteur, l'arité
- des listes, tableaux, fichiers : nombre d'éléments
- des mots : longueur

Comme on exprime la complexité en fonction de la taille de l'instance, il faut la préciser sur le meilleur (raisonnable ou naturel) codage des instances.

2.3.3 Coût des opérations

- Déterminer la complexité d'un algorithme c'est "compter" le nombre d'instructions (opérations) fondamentales qui seront effectuées.
- C'est la nature du problème de calcul qui fait que certaines opérations deviennent plus fondamentales que d'autres dans un algorithme. Ces sont les opérations les plus coûteuses (on dit FLOPS) pour le problème de calcul en question : Racine carrée, Log, Exp, Addition réelle …
- Le nombre d'opérations fondamentales intervient alors principalement dans l'analyse de la complexité de l'algorithme.

Avec un peu d'habitude, on les repère :

Tableau 2.1 : Exemples d'opérations fondamentales

Problème	Opérations fondamentales
Recherche d'un élément dans une liste, un tableau, un arbre …	Comparaisons d'éléments
Tri d'une liste, d'un tableau, d'un fichier …	Comparaisons Déplacements
Multiplication de polynômes, de matrices, de grands entiers	Additions Multiplications

2.3.4 Evaluation des coûts

Evaluation des coûts en séquentiel

Dans un programme strictement séquentiel, les "boucles" sont disjointes ou emboitées : il n'y a pas de récursivité. Les temps d'exécution s'additionnent.

- **Conditionnelle** (si C alors J sinon K)

$$T(n) = T_C(n) + \max\{T_J(n), T_K(n)\} \quad (2.1)$$

- **Itération bornée** (pour i de j à k faire B)

$$\mathrm{T(n)} = (\mathrm{k} - \mathrm{j} + 1) \cdot (\mathrm{T_{entete}(n)} + \mathrm{T_B(n)}) + \mathrm{T_{entete}(n)} \quad (2.2)$$

  entête est mis pour l'affectation de l'indice de boucle et le test de continuation. Le cas des boucles imbriquées se déduit de cette formule.

- **Itération non bornée** (tant que C faire B)

$$\mathrm{T(n)} = \#\mathrm{boucles} \cdot (\mathrm{T_C(n)} + \mathrm{T_B(n)}) + \mathrm{T_C(n)} \quad (2.3)$$

- **Itération non bornée** (répéter B jusqu'à C)

$$\mathrm{T(n)} = \#\mathrm{boucles} \cdot (\mathrm{T_B(n)} + \mathrm{T_C(n)}) \quad (2.4)$$

  Le nombre d'itérations #boucles s'évalue inductivement.

- **Appel de procédures**

  On peut ordonner les procédures de sorte que la i-ième ait sa complexité qui ne dépende que des procédures $j < i$.

Exercice interactif

```
Pour i ← 1 à n Faire
    instruction 1
    Si i est pair Alors
        instruction 2
    Sinon
        instruction 3
        instruction 4
        instruction 5
    FinSi
FinPour
```

Pour ce cas de figure, on pourrait compter n instructions, mais d'autres informations sont disponibles sur la condition donc la complexité sera de $n + \frac{2+4}{2} n$.

Evaluation des coûts en récursif

- Les algorithmes du type "*diviser pour régner*" donnent lieu à des programmes *récursifs*.
  Comment évaluer leur coût ? Il faut trouver :
  - la relation de récurrence associée
  - la base de la récurrence, en examinant les *cas d'arrêt* de la récursion
  - et aussi la solution de cette équation
- Techniques utilisées : résolution des équations de récurrence ou de partition ou par les séries génératrices.
- *Exemple*. (cf. TD)
  On va évaluer le temps $T(n)$ de la recherche dichotomique dans un tableau -trié- de n éléments à

$$n \geq 1 : T(n) = 1 + \log_2(n) \tag{2.5}$$

2.4 Mesures et notations

2.4.1 Notations de la complexité

Le coût en temps d'un algorithme varie évidemment avec la taille n de la donnée x, mais peut aussi varier sur les différentes données de même taille n.

Exemple. Recherche séquentielle

- Le nombre de comparaison réalisées varie de 1 à n

But : évaluer le coût d'un algorithme, selon certains critères et en fonction de la taille n des données. En particulier :

- Coût dans le cas le ***plus défavorable***
- Coût ***moyen***
- Coût dans le ***meilleur des cas***

Coût dans le cas le pire

Le coût d'un algorithme dans le cas le ***plus défavorable*** ou dans le cas ***le pire*** est par définition le maximum de coûts, sur toutes les entrées de taille n :

$$C(n) = \max_{|x|=n} C(n) \tag{2.6}$$

(On note $|x|$ la taille de x)

Exemple. Le cas pire pour le tri par insertion est quand le tableau est trie dans l'ordre inverse, ce coût est : $n(n+1)/2$ (nombre de déplacements effectuées).

Coût moyen

Si l'on pense que le cas le plus défavorable ne se présente que rarement, on est plutôt intéressé par le coût moyen de l'algorithme. On suppose que l'on connaisse une distribution de probabilités sur les données de taille n. Si $p(x)$ est la probabilité de la donnée x, le coût moyen $\gamma(n)$ d'un algorithme sur les données de taille n est par définition :

$$\gamma(n) = \sum_{|x|=n} p(x) \cdot c(x) \tag{2.7}$$

Le plus souvent, on suppose que la distribution est uniforme, c.-à-d. que $p(x\ ) = 1/T(n)$, où $T(n)$ est le nombre de données de taille n.

Exemple. Le coût moyen de l'algorithme de tri par insertion, est $n(n\ +1)/4$.

Exemple. Recherche séquentielle (RS)

```
int index_of_value (double v[], int n, int x)
{
      int j = 0;
      while (j < n)
      {
            if (v[j] == x) return j;
            j = j + 1;
      }
      return -1;
}
```

Opération de base : comparaison de la valeur cherchée x avec un élément du vecteur v.

Complexité en pire cas de *RS* : $max_{RS}(n) = n$

Complexité en moyenne de RS :

On suppose que :

- Tous les éléments sont distincts
- $p[x \in v] = q$
- Si $x \in v$ alors toutes les places sont équiprobables

Pour $1 \leq i \leq n$ soit

$$I_i = \{(v, x), x \in v\} \tag{2.8a}$$

Et

$$I_{n+1} = \{(v, x), x \notin v\} \tag{2.8b}$$

On a :

$$p[I_i] = \frac{q}{n} \quad \text{pour} \quad 1 \leq i \leq n \quad \text{et} \quad C_{RS}(I_i) = i \tag{2.9a}$$

Et

$$p[I_{n+1}] = 1 - q \quad \text{et} \quad C_{RS}(I_{n+1}) = n \tag{2.9b}$$

On calcule la complexité moyenne

$$\begin{aligned} moy_{RS}(n) &= \sum_{i=1}^{n+1} p[I_i] \cdot C_{RS}(I_i) \\ &= \sum_{i=1}^{n} \frac{q}{n}(i) + (1-q) \cdot n \\ &= \frac{q}{n} \sum_{i=1}^{n}(i) + (1-q) \cdot n \\ &= \frac{q}{n} \cdot \frac{n(n+1)}{2} + (1-q) \cdot n \\ &= \left(1 - \frac{q}{2}\right) \cdot n + \frac{q}{2} \end{aligned} \tag{2.10}$$

- Si $q = 1$ alors $Moy_{RS}(n) = \frac{n+1}{2}$
- Si $q = \frac{1}{2}$ alors $Moy_{RS}(n) = \frac{3n+1}{4}$

2.4.2 Notations utilises : Notations de Landau

Evaluer la complexité d'un algorithme consiste à donner l'ordre de grandeur du nombre d'opérations qu'il effectue lorsque la taille du problème qu'il résout augmente. Ce qui permet de comparer le taux d'accroissement de différentes fonctions qui mesurent les performances d'algorithmes. On parle ainsi d'algorithme linéaire, quadratique, logarithmique, etc.

Moyen commode : **Notations de Landau**

- Notation O : donne une **majoration** de l'ordre de grandeur
- Notation Ω : donne une **minoration** de l'ordre de grandeur
- Notation Θ : donne **deux bornes** sur l'ordre de grandeur

Notation *O*

On dit que f -positive- est ***asymptotiquement majorée*** (ou dominée) par g et on écrit :

$$f \in O(g) \quad \text{ou} \quad f = O(g) \tag{2.11a}$$

quand il existe une constante $k > 0$, telle que pour un n assez grand, on a :

$$f(n) \leq k \cdot g(n) \tag{2.11b}$$

Exemple.

$$n \in O(n^2),\ \frac{\ln n}{n} \in O(1),\ x+1 \in O(x) \tag{2.12}$$

La notation $f = O(g)$ est scabreuse car elle ne dénote pas une égalité mais plutôt l'appartenance de f à la classe des fonctions en $O(g)$.

Exemple. $4n^2 + n = O(n^2)$ et $n^2 - 3 = O(n^2)$ sans que l'on ait $4n^2 + n = n^2 - 3$ à partir d'un n assez grand.

Exercice accompagné :

Soit $P(x) = a_0x^k + a_1x^{k-1}+\ldots+ a_{k-1}x^1 + a_k$.

Montrer que $P(x) \in O(x^k)$ au voisinage de l'infini.

Notation Ω

Symétriquement, on dit que f est ***asymptotiquement minorée*** par g et on écrit

$$f \in \Omega(g) \ \text{ou}\ f = \Omega(g) \tag{2.13a}$$

quand il existe une constante $k > 0$, telle que pour un n assez grand, on a :

$$f(n) \geq k \cdot g(n) \tag{2.13b}$$

En d'autres termes, on a

$$f \in \Omega(g) \Leftrightarrow g = O(f) \tag{2.13c}$$

Notation Θ

On dit que f est du ***même ordre de grandeur*** que g et on écrit

$$f \in \Theta(g) \ \text{ou}\ f = \Theta(g) \tag{2.14a}$$

quand il existe deux constantes $k_1, k_2 > 0$, telle que pour un n assez grand, on a

$$k_1 \cdot g(n) \leq f(n) \leq k_2 \cdot g(n) \tag{2.14b}$$

En d'autres termes, on a

$$\Theta(g) = O(g) \cap \Omega(g) \tag{2.14c}$$

Exemple. Pour tout $a, b > 0$ on a

$$\log_a n = \Theta(\log_b n) \tag{2.14d}$$

puisque $\log_a n = \log_a b \log_b n$.

Notations de Landau : Propriétés

- Réflexivité
  - $g = O(g)$

- $g = \Theta(g)$
- Symétrie
  - $f = \Theta(g) \quad ssi \quad g = \Theta(f)$
- Transitivité
  - $f = O(g) \quad et \quad g = O(h) \quad alors \quad f = O(h)$
  - $f = \Theta(g) \quad et \quad g = \Theta(h) \quad alors \quad f = \Theta(h)$
- Produit par un scalaire
  - $c > 0, \quad cO(g) = O(g)$
- Somme et produit de fonctions
  - $O(f) + O(g) = O(\max(f, g))$
  - $O(f)O(g) = O(fg)$

Ordres de grandeurs : Comparaison asymptotique

Tableau 2.2 : Comparaison asymptotique

$$\boldsymbol{log(n) \ll \sqrt{n} \ll n \ll nlog(n) \ll n^2 \ll n^3 \ll 2^n \ll exp(n) \ll n! \ll n^n \ll 2^{2^n}}$$

$\boldsymbol{O(1)}$	**Constant**
$\boldsymbol{O(\log(n))}$	Logarithmique
$\boldsymbol{O(n)}$	Linéaire
$\boldsymbol{O(nlog(n))}$	$nlog(n)$
$\boldsymbol{O(n^2)}$	Quadratique
$\boldsymbol{O(n^3)}$	Cubique
$\boldsymbol{O(2^n)}$	Exponentiel

Algorithmes Polynomiaux vs. Exponentiels

- Complexité **polynomiale** → souvent réalisable
  $\exists k > 0, f(n) \in O(n^k)$.
- Complexité exponentielle → en général irréalisable
  $\exists b > 1, f(n) \in O(b^n)$.
- Complexité ***doublement exponentielle***
  Par exemple : $f(n) = 2^{2^n}$.
- Complexité ***sous exponentielle***
  Par exemple : $f(n) = 2^{\sqrt{n}}$.

Ordres de grandeur : remarques

Tableau 2.3 : Ordres de grandeur

	2	16	64	256
$\log\log n$	0	2	2.58	3
$\log n$	1	4	6	8
n	2	16	64	256
$n\log n$	2	64	384	2048
n^2	4	256	4096	65536
2^n	4	65536	1.84467e+19	1.15792e+77
$n!$	2	2.0923e+13	1.26887e+89	8.57539e+506
n^n	4	1.84467e+19	3.9402e+115	3.2317e+616
2^{n^2}	16	1.15792e+77	1.04438e+1233	2.0035e+19728

Méga = $10^6(2^{20})$, Géga = $10^9(2^{30})$, Tera = $10^{12}(2^{40})$, Péta = $10^{15}(2^{50})$
4 Ghz pendant 1 an = $1{,}26 \times 10^{17}$
4 Ghz pendant 4 Milliards d'années = 5×10^{26}

Chapitre 3

Algorithmes de tri

3.1 Présentation

L'objectif de cette partie de cours est d'illustrer des concepts vus précédemment :

- Complexité des algorithmes
- Récurrence et complexité
- Stratégie "diviser pour régner"

à travers l'étude des algorithmes de tri.

De l'importance du tri

Le tri est l'opération de base sur des données.

- Il est intéressant de trier pour faire plus facilement des recherches : par exemple ordre alphabétique, classement par date, par taille, etc.
- Souvent grande quantité d'information à trier, donc la performance est très importante.
- Dans beaucoup d'algorithmes il faut maintenir de manière dynamique des vecteurs ou listes triés, pour trouver rapidement les plus petits et plus grands éléments.
- Ces méthodes concernent beaucoup d'autres algorithmes.

Dans ce cours, on présente trois algorithmes de tri dits simples, qui sont :

1. Tri bulle (*Bubble sort*)
2. Tri par sélection (*Selection sort*)
3. Tri par insertion (*Insertion sort*)

Et deux algorithmes de tri rapides :

4. Tri fusion (*mergesort*)
5. Tri rapide (*quicksort*)

Le problème du tri considéré dans ce cours est défini de la manière suivante :

Entrée : un tableau de n nombres $T = \{a_1, a_2, \dots, a_n\}$

Sortie : le tableau trié par ordre croissant $T = \{a'_1, a'_2, \dots, a'_n\}$ telle que $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

3.2 Tri bulle

Principe

Comme des bulles qui remontent à la surface d'un liquide :

- Faire remonter la plus grande valeur par permutations successives en fin de tableau
- Puis la deuxième
- … etc.

Exemple

Entrée : 101 115 30 63 47 20
Suite d'échanges pour la première itération
Echange 1 : 101 30 115 63 47 20
Echange 2 : 101 30 63 115 47 20
Echange 3 : 101 30 63 47 115 20
Echange 4 : 101 30 63 47 20 **115**
Suite d'échanges la deuxième itération
Echange 1 : 30 101 63 47 20 **115**
Echange 2 : 30 63 101 47 20 **115**
Echange 3 : 30 63 47 101 20 **115**
Echange 4 : 30 63 47 20 **101** **115**

Suite d'échanges pour la troisième itération
Echange 1 : 30 47 63 20 **101** **115**
Echange 2 : 30 47 20 **63** **101** **115**
Suite d'échanges pour la quatrième itération
Echange 1 : 30 20 **47** **63** **101** **115**
Suite d'échanges pour la cinquième itération
Echange 1 : 20 **30** **47** **63** **101** **115**
Sortie : **20** **30** **47** **63** **101** **115**

Algorithme, réalisation en C :

```c
// Fonction de tri à bulle de l'élément le plus petit
void triBulle (float t[], int n)
{
      int i, k;
      // Boucle d'itérations de l'algorithme
      for (i = 1; i <= n - 1; i++)
            // Boucle pour remonter l'élément i à la fin du tableau
            for (k = i - 1; k < n - i; k++)
                  if (t[k] > t[k + 1])
                        // Permutation de 2 éléments successives
                        swap (t, k, k + 1);
}

// Une fonction auxiliaire pour permuter deux valeurs, se situent
// dans les positions i et j, d'un tableau t
void swap (float t[], int i, int j)
{
      float help;
      help = t[i];
      t[i] = t[j];
      t[j] = help;
}
```

Invariant

Après chaque itération de l'algorithme, $i = 1, 2, ... n - 1$, les i plus grands éléments du tableau sont bien placés (à la fin du tableau).

Complexité

- $\frac{n(n-1)}{4}$ échanges en moyenne et le double dans le pire des cas (quand les valeurs se trouvent dans un ordre décroissant)
- $\frac{n(n-1)}{2}$ comparaisons dans tous le cas

L'algorithme de tri à bulle est en $O(n^2)$

3.3 Tri par sélection

Principe

C'est la méthode de tri le plus simple. Pour trier un tableau, on recherche l'élément minimum du tableau, échange cet élément avec le premier élément du tableau, et recommence sur la fin du tableau où l'on a ajouté l'élément qui se trouvait en première place. Les opérations de sélection de l'élément le plus petit et de son placement à sa place définitive sont répétées jusqu'à ce que tous les éléments soient classés.

Exemple

Entrée : 101 115 30 63 47 20
Sélection de 20
Placement : **20** 115 30 63 47 101
Sélection de 30
Placement : **20 30** 115 63 47 101
Sélection de 47
Placement : **20 30 47** 63 115 101
Sélection de 63
Placement : **20 30 47 63** 115 101
Sélection de 101
Placement : **20 30 47 63 101** 115
Sortie : **20 30 47 63 101 115**

Algorithme, réalisation en C :

```
// Fonction de tri par sélection de l'élément le plus petit
void triSelection (float t[], int n)
{
	int i, j, i_min;
	// Boucle d'itérations de l'algorithme
	for (i = 1; i <= n - 1; i++)
	{
		// Sélection de l'élément le plus petit dans la tranche t[i, n]
		min = i - 1;
		for (j = i; j < n; j++)
			if (t[j] < t[i_min])
				i_min = j;
		// permutation de 2 éléments
		if (i_min > i) swap (t, i_min, i);
	}
}
```

Invariant

A la fin d'un pas d'exécution de la boucle, $i = 1, 2, \ldots n-1$, les i plus petits éléments du tableau sont à leur place définitive (au de but du tableau).

Complexité

- Cas pire (quand les valeurs sont classées dans un ordre décroissant) : $\frac{n(n-1)}{2}$ affectations et $n-1$ échanges d'éléments
- Dans tous le cas : $\frac{n(n-1)}{2}$ comparaisons

L'algorithme de tri par sélection est en $O(n^2)$

Le tri par sélection est intéressant pour trier de gros fichiers avec de petites clés.

3.4 Tri par insertion

Principe

On peut décrire le tri par insertion de la manière suivante :

- On fait $n-1$ itérations, $i = 1, 2, .., n-1$.
- Après l'itération d'algorithme $(i - 1)$ les premiers $(i - 1)$ éléments du tableau sont triés, mais ils ne sont pas forcément à leur place définitive.
- A l'itération i on insère d'une manière séquentielle le i-ème élément du tableau parmi les premiers $(i - 1)$ éléments.

Exemple :

Input	: 101	115	30	63	47	20
Itération 1	: 101	115	30	63	47	20
Itération 2	: 30	101	115	63	47	20
Itération 3	: 30	63	101	115	47	20
Itération 4	: 30	47	63	101	115	20
Itération 5	: 20	30	47	63	101	115
Output	: **20**	**30**	**47**	**63**	**101**	**115**

Algorithme, réalisation en C :

```
// Fonction de tri par insertion de l'élément le plus petit
void triInsertion (float t[], int n)
{
      int i, k ;
      // Boucle d'itérations de l'algorithme
      for (i = 1; i < n; i++)
      {
           // Boucle d'insertion incrémentale
            k = i;
            while (k > 0 && t[k] < t[k - 1]) {
                  swap (t, k, k - 1); k--;
            }
      }
}
```

Invariant

A la fin d'un pas d'exécution de la boucle de l'algorithme, les i premiers éléments sont tries, mais ils ne sont pas forcément à leur place.

Complexité

- Cas pire : $\frac{n(n-1)}{2}$ ***déplacements*** d'éléments (quand les valeurs initiales sont classées dans un ordre décroissant)
- Cas moyen : environ la moitié

L'algorithme de tri par insertion est en $O(n^2)$ (pire de cas).

Il est linéaire si les valeurs sont presque triées

Pour finir sur les algorithmes de tri simples

On peut réaliser ces trois algorithmes simples de manière récursive. Voici une réalisation de tri par insertion :

```
void triInsertionRec (float t[], int n)
{
      if (n > 1)
      {
            triInsertionRec (t, n - 1);
            // insérer t[n - 1] à sa place dans la suite triée obtenue
            // à programmer ...
      }
}
```

3.5 Tri fusion

Principe

Le tri-fusion se fonde sur la stratégie "diviser pour régner".

1. On divise le tableau de départ de n éléments en deux sous-tableaux, ayant respectivement $\lfloor n/2 \rfloor$ et $\lceil n/2 \rceil$ éléments.
2. Un appel récursif est réalisé pour chacun de sous-tableaux.
3. On produit un tableau trié en ***interclassant*** tous les éléments des deux sous-tableaux triés.

Algorithme, réalisation en C :

```
// Fonction de tri fusion
void triFusion (float T[], int g, int d)
// g et d sont les indices gauche et droite des éléments à trier
{
      int m;
      if (g < d)
      {
            // Diviser le tableau initial en deux sous tableaux
            m = (g + d - 1) / 2;
```

```
            // Appel récursif pour trier les deux sous-tableaux
            triFusion (T, g, m);
            triFusion (T, m + 1, d);
            // Interclassement des deux sous-tableaux triés
            interclasser (T, g, m, d);
      }
}
```

Interclassement

La **complexité** de l'algorithme de tri-fusion dépend de l'algorithme choisi pour réaliser l'interclassement. Etant donnée deux tableaux ou listes triés ayant respectivement n_1 et n_2 éléments, il est possible de réaliser leur interclassement dans un tableau résultat ayant $n_1 + n_2$ éléments en temps linaire $O(n_1 + n_2)$ (voir TD). Dans le tri-fusion on peut profiter du fait que les deux listes (sous-tableaux) triés à interclasser se trouvent côte-à-côté dans le tableau initial T pour construire un algorithme d'interclassement plus raffiné que celui du cas général mais qui a la même complexité linéaire. La procédure interclasser ci-dessous recopie dans la partie gauche du tableau de travail R le sous-tableau gauche de T et dans la partie droite le sous-tableau droite inversé. Ce placement astucieux des éléments permet alors de replacer dans T les $n_1 + n_2$ éléments du plus petit au plus grand en répétant $n_1 + n_2$ fois la comparaison des deux éléments courants dans les deux listes. La première liste occupe initialement dans T les positions de g à m et la deuxième liste les positions de $m + 1$ à d.

```
// Fonction pour l'interclassent dans tri-fusion
void interclasser (float T[], int g, int m, int d)
{
      float R[n];
      int i, j, k;
      // Recopie de la première liste dans un tableau de travail R
      for (i = g; i <= m; i++)
            R[i] = T[i];
      // Recopie de la deuxième liste inversée dans R
      for (j = m + 1; j <= d; j++)
            R[j] = T[m + d + 1 - j];
      // Replacement de tous les éléments dans le tableau initial
      i = g; j = d;
      for (k = g; k <= d; k++)
      {
            if (R[i] < R[j])
                  T[k] = R[i], i++;
            else
                  T[k] = R[j], j--;
      }
}
```

La figure ci-dessous illustre l'arborescence des appels récursifs du tri-fusion et de la procédure interclasser pour $n = 8$; on montre les indices de gauche g et de droite d de chaque appel.

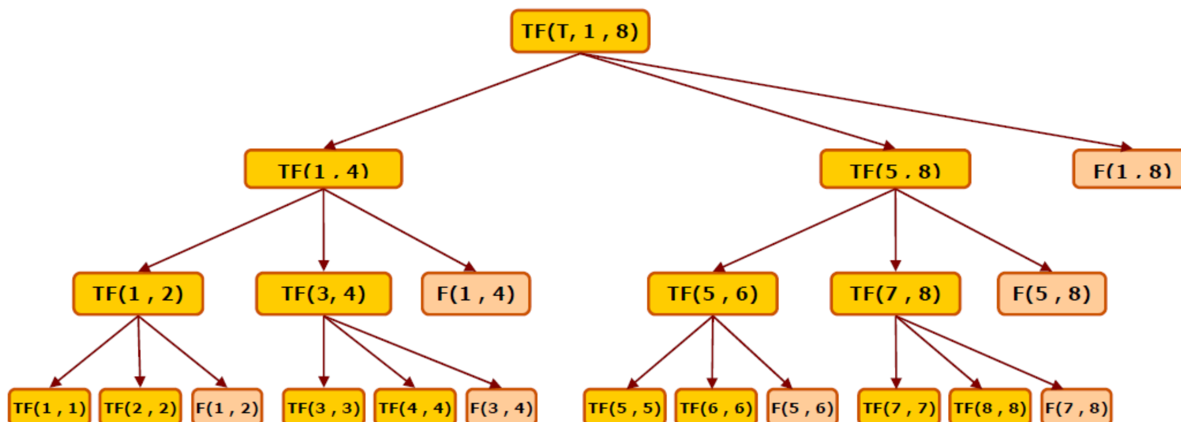


Fig. 3.1 : Illustration de l'arborescence des appels récursifs du tri-fusion

Complexité

Pour simplifier, on suppose que le tableau est de taille $n = 2^k$:

$$c(n) = 2 \times c(n/2) + tfusion(n)$$

où $tfusion$: le coût de l'algorithme d'interclassement

$$tfusion(n) = n - 1 \quad \text{Comparaisons}$$

- Solution asymptotique (récurrences de partition) est : $c(n) = \boldsymbol{\Theta(n \log n)}$
- Solution exacte est : $c(n) = n \log n + 1 - n$

3.6 Tri rapide '*quicksort*'

Principe et implémentation

Le tri rapide utilise le principe "diviser pour régner". Il comprenant deux phases. La première phase transforme le tableau (ou le sous tableau) de départ (de n éléments) en deux listes, *gauche* et *droite* notées G et D respectivement, séparées par un élément appelé le *pivot* noté π. Si la taille de G est $g < n - 1$, la taille de D est $d = n - 1 - g < n - 1$. De plus, tous les éléments de G (resp. D) sont inférieurs (resp. supérieurs) ou égaux à π. Notons aussi que G ou/et D peuvent être vides. En d'autres termes, après cette première phase, le pivot est bien placé dans le tableau (implémentation sans tableau auxiliaire) et les tailles de deux listes sont strictement inferieurs à n. La seconde phase du tri rapide consiste en deux appels récursifs pour les deux parties G et D. Remarquons que pas besoin de "fusionner" après les deux sous-tableaux triés, comme c'est le cas dans le tri-fusion.

Le cas de base de récurrence du tri-rapide (i.e., un appel terminal) est si la partie (G ou D) contient un seul élément ou elle est vide, $n \leq 1$. Dans ce cas, l'algorithme n'effectue aucune comparaison d'éléments. Le code ci-dessous donne une implémentation du tri-rapide sans utilisation d'un tableau auxiliaire. Le code avant les appels récursifs correspond à la procédure de ***pivotage***, qui choisit le pivot puis réorganise le tableau en plaçant les éléments qui sont inférieurs ou égaux au pivot à gauche et les éléments qui sont supérieurs au pivot à droite ; le pivot est mis au milieu.

Dans cette implémentation, le *premier élément du tableau* est choisi comme pivot. Ce choix n'est pas le seul possible ; on peut choisir parmi tous les éléments du tableau en l'absence d'hypothèses supplémentaires sur les données dans le tableau. Ce choix, du premier élément comme le pivot, présente l'avantage d'être simple à implémenter.

Dans cette implémentation, la réorganisation du tableau $v[left, right]$ s'est faite de manière itérative (boucle for dans le code) où à chaque itération on déplace l'élément de l'indice courant au début du tableau, avant le pivot, si sa valeur est inférieure à la valeur du pivot. Un couple indices, k et $last$, est utilisé pour contrôler la permutation, i.e., un couple inversé du tableau par rapport au pivot.

```
void quickSort(int v[], int left, int right)
/*
     FUNCTION:      fonction récursive (quicksort) pour trier un tableau
     INPUT:         un tableau, deux indices
     OUTPUT:        néant
     (SIDE)EFFECTS : éléments dans positions i et j du tab. sont permutés
*/
{
      int k, last;

      if (left >= right)
            return;

      last = left;

      for (k = left + 1; k <= right; k++)
      {
            if (v[k] < v[left])
            {
                  swap(v, ++last, k);
                  /* fonction swap permute 2 éléments d'un tableau
                    en passant leur indices */
            }
      }
      swap(v, left, last);
      /* appels récursifs pour les deux sous-tableau */
      quickSort (v, left, last - 1);
      quickSort (v, last + 1, right);
}
```

Complexité de tri rapide

Complexité dans le pire des cas

Le cas le plus défavorable est celui où le pivot choisi est toujours le plus petit élément du tableau. Il est attient lorsque le tableau initial est trié. Dans ce cas, pour chaque appel récursif, la partition donne une liste gauche G vide et une liste droite D contenant les $n-1$ autres éléments. Le nombre de comparaisons effectuées pour la réorganisation du tableau est $n-1$. D'où l'équation de récurrence :

$$L_n = L_{n-1} + n - 1 \qquad L_0 = 0 \tag{3.1}$$

$$\mathrm{L_n = n(n-1)/2} \tag{3.2}$$

La complexité du tri-rapide est donc $O(n^2)$ dans le pire cas.

Le cas le plus favorable de tri-rapide

Le cas le plus favorable est celui où le pivot choisi est l'élément médian du tableau. Dans ce cas, le tableau est partitionné en deux parties de même longueur $n/2$. D'où l'équation de récurrence de partition :

$$\mathrm{L(n)} = 2\mathrm{L}\left(\frac{\mathrm{n}}{2}\right) + \mathrm{n} - 1 \qquad \mathrm{L}(1) = 1 \tag{3.3}$$

La solution de cette équation est : $O(n \log n)$

Complexité en moyenne

On peut envisager la notion de moyenne en considérant les $n!$ permutations (ou ordres) possibles de n nombres du tableau et en faisant la moyenne du nombre de comparaisons.

En moyenne, $n - 1$ comparaisons sont toujours faites pour partitionner le tableau. Intéressons-nous à la position i du pivot. Chaque position $i = 1, 2, \dots, n$ à une probabilité de $1/n$ d'être choisie, donc :

$$\mathrm{F_n} = \mathrm{n} + 1 + \frac{1}{\mathrm{n}}\sum_{\mathrm{i}=1}^{\mathrm{n}}(\mathrm{F_{i-1}} + \mathrm{F_{n-i}}) \tag{3.4}$$

Mais :

$$\sum_{\mathrm{i}=1}^{\mathrm{n}} \mathrm{F_{i-1}} = \mathrm{F_0} + \mathrm{F_1} + \cdots + \mathrm{F_{n-1}} \tag{3.5}$$

$$\sum_{\mathrm{i}=1}^{\mathrm{n}} \mathrm{F_{n-i}} = \mathrm{F_{n-1}} + \mathrm{F_{n-2}} + \cdots + \mathrm{F_0} \tag{3.6}$$

en multipliant par n : $\boldsymbol{nF_n = n(n-1) + 2\sum_{i=0}^{n-1} F_i}$

$$\mathrm{nF_n} = \mathrm{n(n+1)} + 2\sum_{\mathrm{i}=1}^{\mathrm{n}-1} \mathrm{F_{i-1}} + 2\mathrm{F_{n-1}} \tag{3.7}$$

$$\mathrm{(n-1)F_{n-1}} = \mathrm{(n-1)n} + 2\sum_{\mathrm{i}=1}^{\mathrm{n}-1} \mathrm{F_{i-1}} \tag{3.8}$$

en soustrayant on obtient :

$$\mathrm{nF_n} = \mathrm{(n+1)F_{n-1}} + 2\mathrm{n} \qquad \mathrm{F_1} = 2 \tag{3.9}$$

En divisant par $n(n+1)$ on obtient :

$$\frac{\mathrm{F_n}}{\mathrm{n+1}} = \frac{\mathrm{F_{n-1}}}{\mathrm{n}} + \frac{2}{\mathrm{n+1}} \tag{3.10}$$

$$\frac{\mathrm{F_n}}{\mathrm{n+1}} = \frac{\mathrm{F_1}}{2} + 2\sum_{\mathrm{i}=3}^{\mathrm{n}+1}\frac{1}{\mathrm{i}} \tag{3.11}$$

$$\mathrm{F_n} = 2(\mathrm{n}+1)(-1 + \sum_{\mathrm{i}=1}^{\mathrm{n}}\frac{1}{\mathrm{i}}) \tag{3.12}$$

on sait que :

$$\sum_{\mathrm{i}=1}^{\mathrm{n}}\frac{1}{\mathrm{i}} = \mathrm{H_n} = \ln \mathrm{n} + 0.57721 \tag{3.13}$$

$$\mathrm{F_n} = 2\mathrm{n}\ln \mathrm{n} - 0.846\mathrm{n} \tag{3.14}$$

Le nombre moyen de comparaisons effectuées par l'algorithme du tri-rapide pour trier une suite de n éléments est de l'ordre de $2n \log n$.

3.7 Complément de cours : Recherche par dichotomie

Une recherche dichotomique consiste à profiter de l'ordre (croissant) des éléments dans un tableau, $v[1, n]$, pour accélérer la recherche en procédant comme suit :

- On considère l'élément figurant au "milieu" du tableau, de l'indice $\frac{n}{2}$. Si la valeur cherchée lui est égale, la recherche est terminée. S'elle lui est inférieure, on en conclut que valeur cherchée peut-être se trouver que dans la première moitié du tableau $v[1, \frac{n}{2}]$; dans le cas contraire, on en conclut qu'elle peut-être se trouver dans la seconde moitié, $v[\frac{n}{2} + 1, n]$.
- On recommence alors l'opération sur la moitié concernée, puis sur la moitié de cette moitié, et ainsi de suite … jusqu'à ce que l'une des conditions suivantes soit vérifiée :
  - On a trouvé la valeur cherchée ; la valeur cherchée est égale l'élément de milieu courant
  - On est sûr qu'elle ne se figure pas dans le tableau ; la partie du tableau concerné par la recherche est vide

On donne ci-dessous une implémentation séquentielle et une implémentation récursive de la recherche dichotomique. Dans les deux fonctions l'indice de milieu est calculé à partir de l'indice de gauche et l'indice de droite de la partie de tableau concerné par la recherche.

```
// Fonction itérative
int RechercheDichotomique(float t[], int n, float x)
      /* x: valeur cherchée */
{
      int gauche;    /* limite gauche de la recherche*/
      int droite;    /* limite droite de la recherche*/
      int milieu;    /* nouvelle limite (droite ou gauche)*/
      int ele_cour;  /* élément courant*/
      int trouve;    /* indicateur élément trouvé/non trouvé*/

      gauche = 0;
      droite = n - 1 ;
      trouve = 0;
      while (gauche <= droite)
      {
            milieu = (gauche + droite) / 2;
            ele_cour = t[milieu];
            if (ele_cour == x) return milieu;
            else if (ele_cour < x)
                  gauche = milieu + 1;
            else
                  droite = milieu - 1;
      }
      return (-1);
}
// Fonction récursive
int RechDichRec(float t[], float x, int g, int d)
{
      int m;
      if (g > d) return 0; /*appel terminal*/

      m = (g + d) / 2;
      if (x == t[m]) return 1;
```

```
      if (x > t[m])
            return RechDichRec(t, x, m + 1, d);
           /*poursuivre la recherche dans la moitié droite*/
      else
            return RechDichRec(t, x, g, m - 1);
           /*poursuivre la recherche dans la moitié gauche*/
}
```

Complexité de la recherche par dichotomie

Le cas le plus favorable est celui où la valeur cherchée est égale à la valeur de l'élément de milieu (de première itération dans la fonction itérative et de premier appel de la fonction récursive). Dans ce cas, l'algorithme fait une seule comparaison puis retourne l'indice de l'existence.

Le cas le plus défavorable est celui où l'algorithme fait toutes les opérations de division du tableau, pour décider que la valeur cherchée ne se figure pas dans le tableau (on arrive à une moitié vide dans la fonction itérative ou à un appel terminal dans la version récursive) ou elle est égale à la valeur de l'élément dont l'indice est celui calculé à partir d'un indice de gauche égale à un indice de droite. Dans ce cas, la recherche par dichotomie fait $\log n + 1$ comparaisons. La complexité de la recherche par dichotomie est donc $O(\log n)$.

En ***moyenne***, l'algorithme fait

$$\frac{1}{\log n+1}\sum_{i=1}^{\log n+1} i = \frac{(\log n+1)(\log n+2)}{2} \qquad (3.15)$$

Chapitre 4

Les arbres

4.1 Introduction

Les structures collectives déjà étudiées sont les structures linéaires :

- Tableaux
  - L'accès aux éléments est indexé, direct ($v[i]$),
  - Le traitement est itératif en utilisant les structures de contrôle répétitives : `for, while` et `do … while`.
- Listes chainées
  - L'accès aux données est séquentiel en utilisant un champ de structure qui contient l'adresses mémoire de la cellule suivante (`l->suivant`),
  - Le traitement est récursif en exploitent le fait que le suivant d'une cellule est une liste : `if(EstListeVide(l->suivant)!= 1)`

Le but de cette partie du cours est de décrire les structures de données ***hiérarchiques*** (ou arborescentes) :

- Arbre général, n-aire : les nœuds ont un nombre quelconque de suivants
  - **Arbre binaire** : chaque nœud a 0, 1 ou 2 suivants
- Traitement récursif : les suivants sont aussi des arbres
- Un arbre n'a pas de cycle, moins général qu'un graphe

4.2 Définitions

Un ***arbre*** représente une collection finie de ***nœuds***, de même type de données.

Définition récursive : Un arbre soit l'arbre vide, soit formé de son premier nœud, appelé **racine** de l'arbre, et 0, 1, 2 ou plusieurs ***sous arbres*** disjoints appelés ***descendants*** de la racine.

Vocabulaire :

- Chaque nœud stocke une information appelé la ***valeur***/***contenu***/***clé*** de nœud
- Les nœuds sans descendant sont appelés ***Feuilles*** (nœuds externes vs. nœuds internes)

- Les descendants d’un nœud sont appelés leur ***fils***
- Chaque nœud est le ***père*** de ses fils
- Les nœuds du même père sont ***frères***

Exemple :

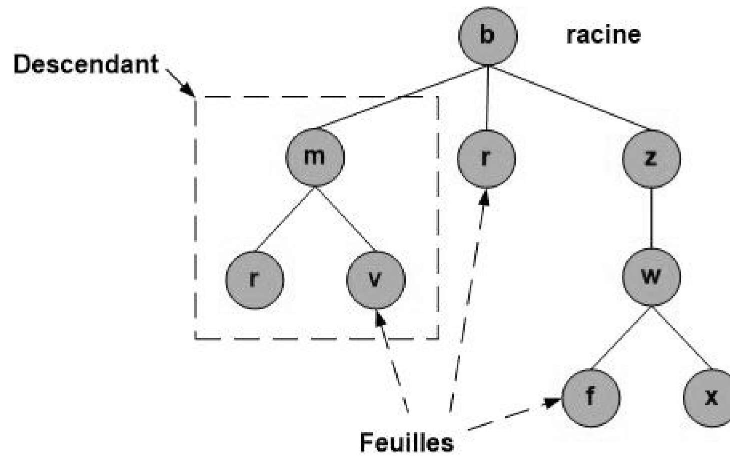


Fig. 4.1 : Représentation graphique d’un arbre

L’arbre de l’exemple stocke de valeurs de type `char`. Le nœud de valeur w est un fils de nœud de valeur z. Les feuilles de l’arbre sont r, v, f, x. Les nœuds de valeurs m, r, et z sont frères, leur père est la racine de l’arbre de valeur b.

4.2.1 Représentation chaînée d’un arbre *n*-aire

Pour implémenter en C un arbre général n-aire, on définit les trois types suivants :

- type des données stockées aux nœuds : `element`
- type des nœuds : `arbre`
- type des sous-arbres immédiats (fils), nombre quelconque de suivants : `liste_d_arbres`

D’un côté, un arbre est déclaré comme un couple : (contenu, `liste_d_arbres`) :

```
typedef struct arbre
{
	element contenu; /* contenu ou clé du noeud */
	liste_d_arbres fils; /* Liste chaînée de sous arbres */
	/* immédiats (fils) du noeud */
} arbre;
```

D’un autre côté, la liste d’arbres (fils) est déclarée comme une liste chaînée :

```
typedef struct maillon
{
	arbre sousArbre;
	struct maillon *frere; /* pointeur vers la cellule suivante */
} maillon, *liste_d_arbre;
```

4.3 Arbre binaire

4.3.1 Définition

Les nœuds d'un ***arbre binaire*** ont chacun au plus deux fils. Un des deux fils est appelé le ***descendant gauche*** et l'autre le ***descendant droit***. Un arbre binaire peut être l'arbre vide.

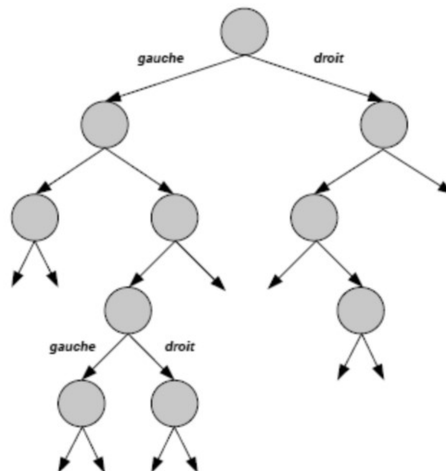


Fig. 4.2 : Illustration d'un arbre binaire

4.3.2 Représentation chaînée d'un arbre binaire

Pour implémenter en C un arbre binaire, on définit les trois types suivants :

- Type des données stockées aux nœuds : `element`
- Type des nœuds : `nœud`
- Type des arbres binaires : `arbre`

Une **relation récursive** entre ces types est : un arbre binaire est un nœud racine. Les descendants d'un nœud sont aussi des arbres binaires (sous-arbres). Les types **nœud** et **arbre** s'utilisent l'un l'autre : les descendants d'un nœud sont des pointeurs de type nœud. On identifie un arbre avec l'adresse de sa racine :

```
typedef struct noeud* arbre;
```

Définition des types

```
typedef char element; /* changer ici type char pour un autre type */
typedef struct noeud* arbre;
typedef struct noeud {
      element contenu;
      arbre gauche;
      arbre droit;
} noeud;
```

Remarque : le type **arbre** n'est pas un type `struct`, est un pointeur sur le type `struct`.

Illustration de la représentation chainée d'un arbre binaire :

- Arbre vide : pointeur sur **NULL**
- Exemple : arbre binaire à 5 nœuds dont les valeurs sont de type **char**

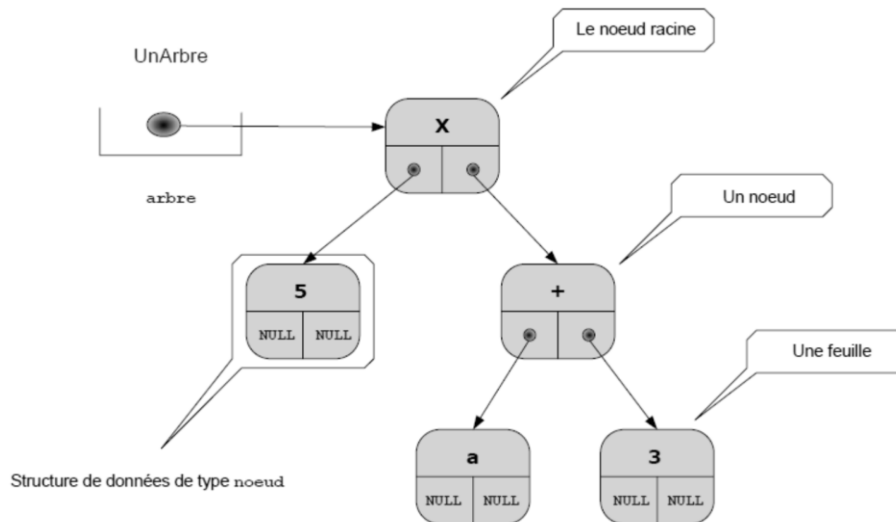


Fig. 4.3 : Illustration de l'implémentation d'un ABR par pointeurs

4.3.3 Opérations sur un arbre binaire

Nous décrivons ici les opérations de base permettant de manipuler les arbres binaires en implémentation par pointeurs. Toutes ces opérations sont dans le type nœud, car

```
typedef struct noeud *arbre;
```

Trois primitives d'accès aux champs sont :

```
arbre SousArbreGauche(arbre a);
```

Renvoie le sous-arbre gauche de l'arbre a; a doit être non vide.

```
arbre SousArbreDroit(arbre a);
```

Renvoie le sous-arbre droit de l'arbre a; a doit être non vide.

```
element Cle(arbre a);
```

Renvoie l'élément (clé) de la racine de l'arbre a ; a doit être non vide.

Traitement récursif

```
int EstFeuille(arbre a);
```

Teste si la racine de l'arbre a est une feuille ; a doit être non vide. Une condition d'arrêt d'un parcours descendant.

```
int EstArbreVide(arbre a);
```

Teste si un arbre a est vide. Une condition d'accès à la clé et aux descendants (gauche et droit).

Opérations de construction et de modification d'arbres

Opérations à définir à l'aide des opérations de base.

```
void ArbreVide(arbre *a);
```
Crée un arbre vide a.

```
arbre FaireFeuille(element x) ;
```
Crée un arbre réduit à un seul nœud qui contient x.

```
void FixerCle(element x, arbre a) ;
```
Remplace le contenu de la racine de a par x ; a doit être non vide.

```
arbre FaireArabre(element x, arbre g, arbre d)
```
Crée un nouveau nœud qui contient x, et retourne l'arbre ayant ce nœud pour racine, et g et d comme des sous-arbres gauche et droit.

```
Void FixerGauche(arbre g, arbre a);
```
Remplace le sous-arbre gauche de l'arbre a par l'arbre g ; a doit être non vide.

```
Void FixerGauche(arbre d, arbre a);
```
Remplace le sous-arbre droit de l'arbre a par l'arbre d ; a doit être non vide.

Ces opérations … à programmer en exercice.

4.3.4 Parcours d'un arbre binaire

Les arbres (structures récursives) permettent de définir des parcours spécifiques. Trois types de parcours sont très utilisés :

- ***Parcours préfixe (préordre ou RGD)*** pour lequel le nœud est placé avant ses nœuds fils dans la liste. Sur l'arbre de l'exemple le résultat de ce parcours est la liste : `abdecfgh`
- ***Parcours infixe (projectif, symétrique ou encore GRD)*** pour lequel le nœud père est placé entre ses descendants dans la liste. Sur l'arbre de l'exemple la liste est : `ddeacgfh`
- ***Parcours postfixe (ordre terminal ou GDR)*** pour lequel les descendants propres d'un nœud père sont placés avant ce nœud dans la liste. Sur l'arbre de l'exemple, la liste est : `debghfca`

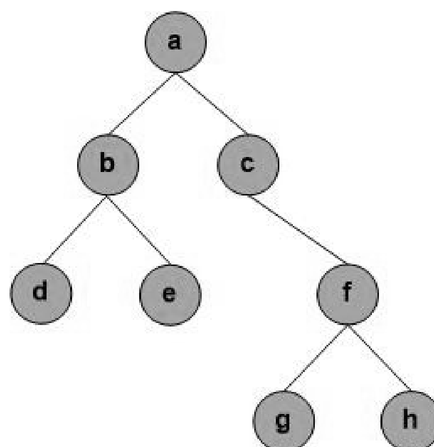


Fig. 4.4 : Arbre exemple pour illustrer les parcours d'un arbre binaire

Programmation en C

```
// Parcours pré-ordre ou préfixé
void ParcoursPreOrdre(arbre a)
{
      while(!EstArbreVide(a))
      {
            printf("%d, ", Cle(a));
            ParcoursPreOrdre(SousArbreGauche(a));
            ParcoursPreOrdre(SousArbreDroit(a));
   }
}

// Parcours in-ordre ou infixé
void parcoursInOrdre(arbre a)
{
      while(!EstArbreVide(a))
      {
            ParcoursPreOrdre(SousArbreGauche(a));
            Printf("%d, ", Cle(a));
            ParcoursPreOrdre(SousArbreDroit(a));
      }
}

// Parcours post-ordre ou postfixé
void parcoursPostOrdre(arbre a)
{
      while(!EstArbreVide(a))
      {
            ParcoursPostOrdre(SousArbreGauche(a));
            ParcoursPostOrdre(SousArbreDroit(a));
            printf("%d, ", Cle(a));
      }
}
```

4.3.5 Arbres binaires particuliers

4.3.5.1 *Arbre binaire complet*

Un ***arbre binaire complet*** (ABC) est un arbre binaire non vide et complet. Chaque nœud dans un ABC soit est une feuille (nœud externe) soit un nœud interne avec deux fils. Voici une définition récursive utile des ABCs :

1. Un arbre réduit à sa racine est un ABC, c'est le premier ABC ;
2. Soient deux ABCs a et b et une valeur v qui ne se figure pas ni dans a ni dans b, l'arbre c qui a la valeur v dans sa racine et de sous-arbre gauche a et de sous-arbre droit b est un ABC.

La figure ci-dessous illustre les premiers ABCs.

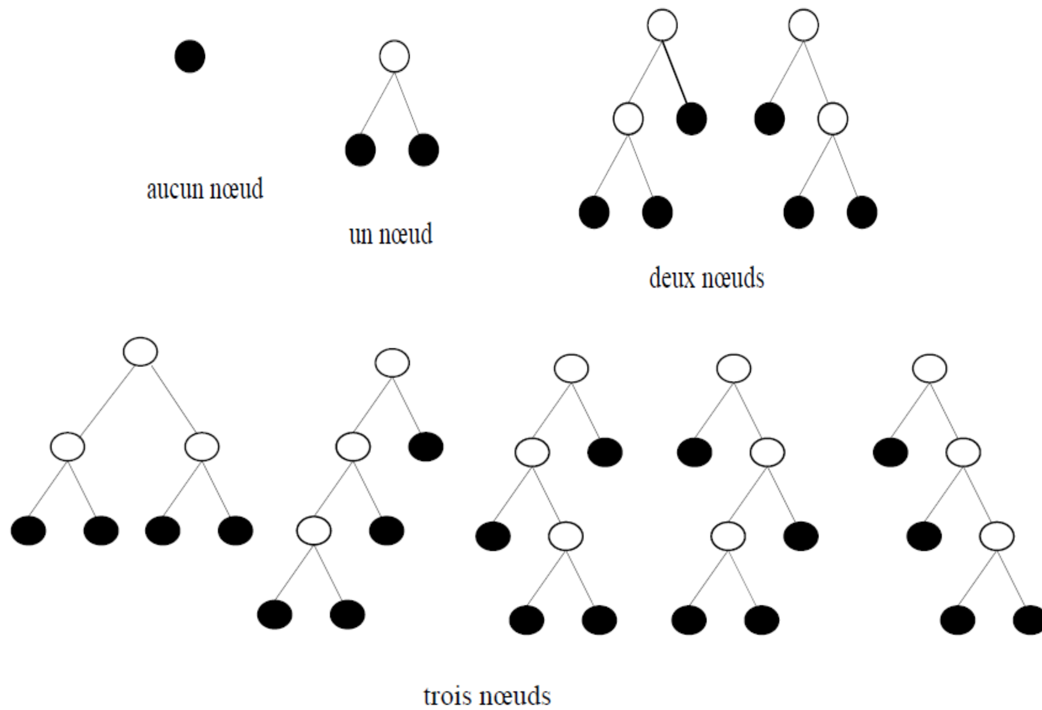


Fig. 4.5 : Illustration des premiers ABCs.

4.3.5.2 *Arbre binaire de recherche*

Un ***arbre binaire de recherche*** (ABR) est un arbre binaire particulier dont chaque nœud est muni d'une clé prise dans un ensemble ordonnée. Un ABR a la propriété fondamentale suivante : pour chaque nœud (sous-arbre), toutes les clés contenues dans son sous-arbre gauche sont inférieures ou égales à la clé du nœud considéré et cette clé est elle-même strictement inférieure aux clés contenues dans le sous-arbre droit. En d'autres termes, un parcours infixe (symétrique), comme décrit dans la section précédente, produit une liste de clés en ordre croissant. Voici une spécification de cette propriété d'ordre :
Pour chaque nœud (sous-arbre) n de l'arbre, on s'assure que :

$$maxCle(n \rightarrow gauche) \leq cle(n) < minCle(n \rightarrow droit) \quad (4.1)$$

Deux conditions doivent être vérifiées pour construire des ABRs :

1. Le type `element`, type de données contenues dans les nœuds, doit être ordonné
2. Tout ajout, toute suppression de nœud doit maintenir la condition (4.1) vérifiée

Exercice interactif

- Vérifier que l'ordre est maintenu
- Ajouter des nouveaux éléments (entiers) en conservant l'ordre

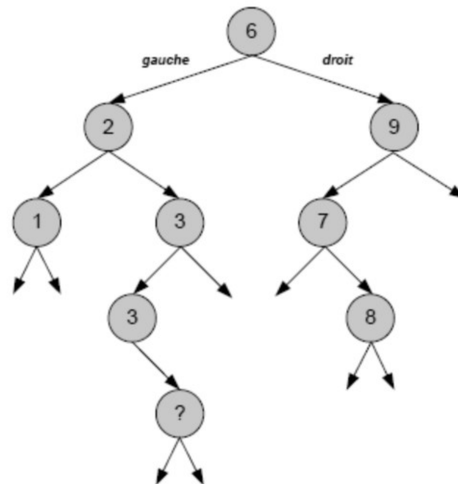


Fig. 4.6 : Arbre de l’exercice interactif

Remarques

- L’insertion de nouvelles valeurs se fait au niveau des feuilles de l’arbre.
- On peut supprimer immédiatement des feuilles.
- Pour la suppression d’un nœud à 1 ou 2 descendants, voir plus loin.

Types et opérations pour les ABRs

- On utilise les mêmes types définis dans la section 3.3.3 pour l’implémentation des arbres binaires (un arbre est un pointeur sur nœud).
- On doit définir les opérations de modification (ajout et suppression) qui préservent la propriété d’ordre, voir plus loin.

Ajout dans un ABR

```
void inserer(element x, arbre *a);
```

L’arbre a contient un nœud supplémentaire de valeur égale à x et l’ordre est conservé après cette insertion. On crée une nouvelle feuille dont la clé est x.

Algorithme d’insertion récursif :

- Appel terminal : si a est vide, on crée une feuille et lui donne x ; a est remplacé par le nouvel arbre (nouvelle feuille)
- Appel récursif
  - Si x est inférieure ou égale à la clé du nœud a, on insère x dans le sous-arbre gauche de l’arbre a
  - Sinon si x est supérieur à la clé du nœud a, on insère x dans le sous-arbre gauche de l’arbre a

Surpression dans un ABR

```
void supprimer(element x, arbre *a);
```

Le nœud n de l’arbre a, dont le contenu est x, est supprimé et l’ordre est conservé sur l’arbre résultat.

Trois cas de nœud à supprimer :

1. Une feuille : il suffit de la supprimer.
2. Un nœud avec un seul fils : on le remplace par son fils unique.
3. Un nœud avec 2 fils : on remplace son contenu par la valeur maximale de son sous-arbre gauche (c'est le nœud le plus à droite dont le contenu précède le contenu du nœud considéré dans l'ordre infixe), et on supprime le nœud qui a cette valeur (appel récursif).
   Symétriquement, on pourrait aussi remplacer le contenu de nœud à supprimer par celui du nœud qui lui succède dans l'ordre infixe (c'est le nœud le plus à gauche dans le sous-arbre droit).

Recherche dans un ABR

```
int rechercher(element x, arbre a);
```

Teste si x figure dans l'arbre a. Elle retourne 1 si x est dans a et 0 sinon.

Un ABR permet une recherche efficace, inspirée par la dichotomie ($O(\log n)$).

Algorithme de recherche récursif :

- Cas d'arrêt avec échec : l'arbre est vide
- Cas d'arrêt avec succès : x est la valeur du nœud racine
- Poursuite de la recherche dans le sous-arbre gauche ou dans le sous-arbre droit, selon le résultat de comparaison de la valeur cherchée avec la clé du nœud courant
- Arrêt de l'algorithme ? Oui, car la taille (ici la hauteur) de l'arbre exploré est un entier naturel qui décroit strictement.

Les opérations sur ABRS : `inserer`, `supprimer`, `rechercher` … à programmer en exercice.

Problème d'équilibrage d'un ABR

Une suite d'opérations d'insertion et de suppression dans un ABR peut *déséquilibrer* l'arbre et rendre les opérations futures moins efficaces (la recherche n'est plus logarithmique avec la taille).

Définition : le sous-arbre gauche d'un ABR équilibré est de même taille que son sous-arbre droit. Pour tout nœud n de l'arbre,

$$|\text{hauteur}(\text{n. gauche}) - \text{hauteur}(\text{n. droite})| \leq 1 \quad (4.2)$$

La fonction *hauteur* exprime la hauteur de l'arbre. La hauteur d'un arbre est la longueur maximale entre sa racine et une de ses feuilles.

Les algorithmes existants pour équilibrer les ABRs sont assez coûteux en termes calcul ; à utiliser de temps en temps.

Chapitre 5

Les graphes

Introduction

Les graphes sont des structures de données plus utilisées pour représenter et étudier les liaisons entres les éléments d'un ensemble d'objets (souvent de même type). Voici quelques exemples de situations modélisées par un graphe :

- Les liaisons entre les nœuds dans un réseau informatique d'un problème de routage,
- Les routes entre différents sites dans un réseau routier,
- Les liens hypertexte entre les pages web,
- Les amitiés entre les personnes dans un réseau social,
- Les relations d'adjacences entre les régions dans une carte géographique,
- Les chevauchements entre les fragments d'ADN d'un problème d'assemblage d'un génome,
- … etc.

Comme les graphes représentent un outil théorique fondamental, le développement des algorithmes les plus efficaces pour réaliser les opérations de base sur les graphes constitue un objectif essentiel de l'algorithmique.

5.1 Définitions

Graphes orientés

Un ***graphe orienté*** est déterminé par la donnée d'un couple $G = (S, A)$, où S est un ensemble fini non vide d'objets (pas nécessairement de même type) appelés ***sommets***, et A est une partie de produit $S \times S$ dont les éléments sont appelés ***arcs***. Les arcs constituent une relation *orientée* entre les sommets. Dans un graphe orienté, la composante x d'un arc $(x, y) \in A$ est dite le sommet d'***origine*** et la composante y est dite le sommet d'***extrémité***. Si $(x, y) \in A$, y est un ***successeur*** de x et x est un ***prédécesseur*** de y. Un arc dont l'origine et l'extrémité sont le même sommet, de la forme (x, x), est appelé une ***boucle***. Un sommet qui n'apparait dans aucun arc de A est un sommet ***isolé***.

Les sommets d'un graphe sont représentés graphiquement par des points du plan. Un arc d'un graphe orienté est représenté par une flèche du sommet d'origine vers le sommet d'extrémité.

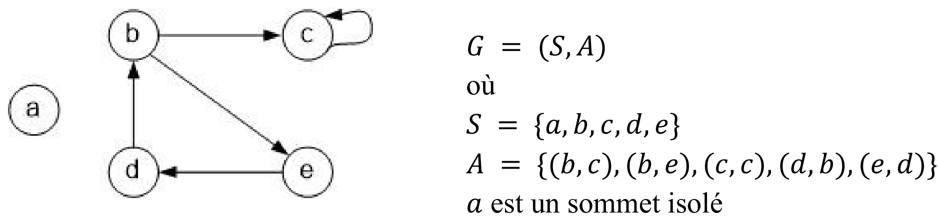


Fig. 5.1 : Un graphe orienté

Graphes non orientés

Un ***graphe non orienté*** est un couple $G = (S, A)$, où S est un ensemble fini non vide d'éléments appelés ***sommets***, et A est un ensemble de paires (couples non ordonnés) d'éléments de S dont les éléments sont appelés ***arrêtes***. La version non orientée d'un graphe orienté est obtenue en suppriment toutes les boucles et en remplaçant chaque arc (x, y) par la paire $\{x, y\}$. De même, étant donné un graphe non orienté sans boucles, sa représentation équivalente par un graphe orienté est obtenue en substituant à chaque arrête $\{x, y\}$ les deux arcs (x, y) et (y, x).

Les arêtes d'un graphe non oriente sont dessinées par des lignes joignant les sommets.

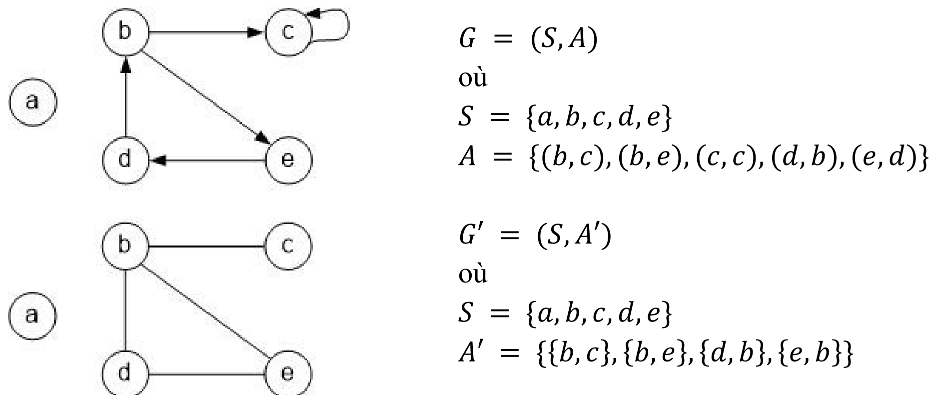


Fig. 5.2 : Un graphe orienté et sa version non orientée

Si (x, y) un arc (respectivement $\{x, y\}$ une arrête) dans un graphe orienté (respectivement non oriente), les sommets x et y sont dits ***adjacents*** ou ***voisins***.

Un arc (x, y) dans un graphe orienté (respectivement une arrête $\{x, y\}$ dans un graphe non orienté) est dit(e) ***incident*** de leur sommet extrémités x et y.

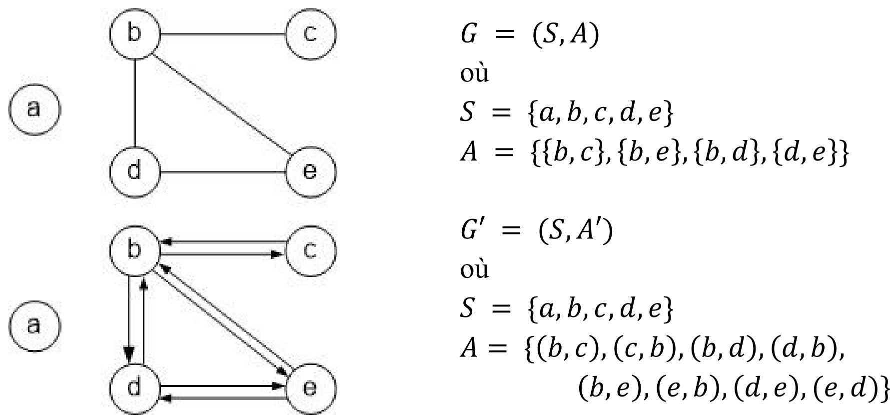


Fig. 5.3 : Un graphe non orienté et sa représentation équivalente par un graphe orienté

Degré d'un sommet ou d'un graphe

Le ***degré d'un sommet*** $s \in S$, noté $d(s)$, dans un graphe $G = (S, A)$ est le nombre d'arcs ou d'arrêtes incidents. Le ***degré d'un graphe,*** noté $d(G)$, est le degré du sommet qui a le degré le plus élevé :

$$d(G) = \max_{s \in S} d(s) \tag{5.1}$$

Sous-graphes

Un ***sous-graphe*** d'un graphe (orienté ou non orienté) $G = (S, A)$ est donné par un couple $G' = (S', A')$, où $S' \subseteq S$, $A' \subseteq A$. Les extrémités des arcs ou des arêtes de A' sont toutes dans S'. On dit que le sous-graphe G' est ***induit*** par S'. Un sous-graphe G' est dit un graphe ***partiel*** de G si $S' = S$.

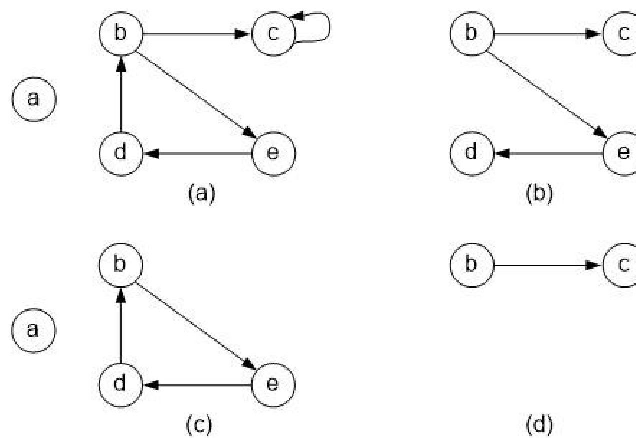


Fig. 5.4 : Sous-graphes. Les graphes en (b), (c) et (d) sont des sous-graphes du graphe orienté en (a).

Connexité

Un graphe non orienté est dit ***connexe*** s'il y a un chemin (une séquence de sommets et d'arcs) reliant n'importe quelle paire de sommets distincts. La connexité s'applique aussi aux graphes orientés si en considérant pas les orientations des arcs.

5.2 Représentations informatiques d'un graphe

Trois implémentations fondamentales d'un graphe orienté $G = (S, A)$ interviennent dans la plupart des problèmes de calcul théoriques et pratiques (problèmes de routage et problèmes d'ordonnancement, etc.), qui sont :

1. La matrice d'adjacence,
2. La matrice d'incidence et
3. La liste des successeurs (ou la liste des voisins dans le cas d'un graphe non orienté)

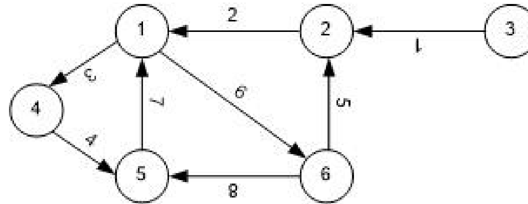


Fig. 5.5 : Un graphe orienté pour l'illustration de trois implémentations

On note n le nombre de sommets et m le nombre d'arcs (ou d'arêtes pour un graphe non orienté).

La **matrice d'adjacence** $adj(G)$ est une matrice binaire carrée de taille $n \times n$, définie comme suit :

$$\mathrm{adj_{ij}} = \begin{cases} 1 & \text{si } (i,j) \in A \\ 0 & \text{sinon} \end{cases} \tag{5.2}$$

où les nombres binaires 0 et 1 sont interprétés selon le problème en considération : présence d'une relation d'adjacence entre deux sommets.

La matrice d'adjacence du graphe orienté de la figure 5.2 est la suivante :

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

Pour un graphe sans boucles, la **matrice d'incidence** « sommets-arcs » $\Delta(G)$ est une matrice de taille $n \times m$, définie par :

$$\Delta_{\mathrm{xa}} = \begin{cases} 1 & \text{si x est l'origine de l'arc a} \\ -1 & \text{si x est l'extrimité de l'arc a} \\ 0 & \text{sinon} \end{cases} \tag{5.3}$$

La matrice d'incidence du graphe orienté de la figure 4.2 est la suivante :

$$\Delta = \begin{pmatrix} 0 & -1 & 1 & 0 & 0 & 1 & -1 & 0 \\ -1 & 1 & 0 & 0 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 1 \end{pmatrix}$$

La **liste des successeurs** est un tableau $(q_1, q_2, \dots, q_n)$ où q_i est un pointeur sur une liste (chainée) des sommets successeurs du sommet i.

Pour un graphe non orienté, la représentation la plus utilisée est la **liste des voisins** donnée par un tableau $(q_1, q_2, \dots, q_n)$ où chaque élément q_i pointe sur la liste (chainée) des sommets voisins du sommet i.

La taille de la liste des successeurs ou des voisins est en $O(n + m)$.

Figure 5.3 montre la liste des successeurs du graphe orienté présenté par Fig. 5.2. Une illustration de la liste des voisins de la version non orienté du même graphe est donnée dans Fig. 5.4.

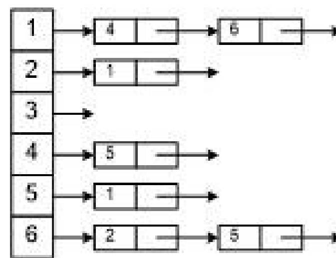


Fig. 5.6 : Implémentation d'un graphe orienté par une liste des successeurs

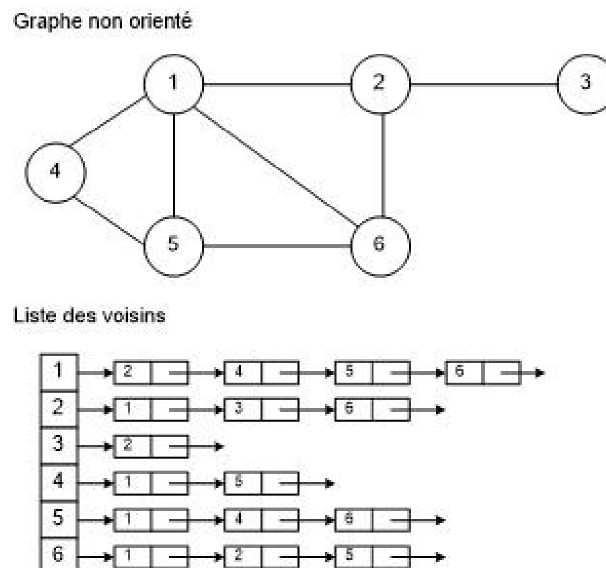


Fig. 5.7 : Implémentation d'un graphe non orienté par une liste des voisins

5.3 Parcours d'un graphe

Dans cette section, nous présentons le parcours d'un graphe non orienté connexe $G = (S, A)$. On parle d'un parcours d'un graphe lorsqu'on passe d'un sommet à un autre on se déplaçant le long d'arêtes et de sommets. Un parcours de G qui commence à partir d'un sommet quelconque $s \in S$ est une liste de tous les sommets L, telle que, chaque sommet de S apparaît une seule fois dans L, et tout sommet de L (sauf le sommet de départ s) est voisin de au moins un sommet placé avant lui dans L.

Nous présentons d'abord un algorithme générique pour parcourir un graphe, puis les deux types de parcours les plus utilisés : le *parcours en profondeur* et le *parcours en largeur*. Nous illustrons les différents parcours sur le graphe de Fig. 4.5 ci-dessous :

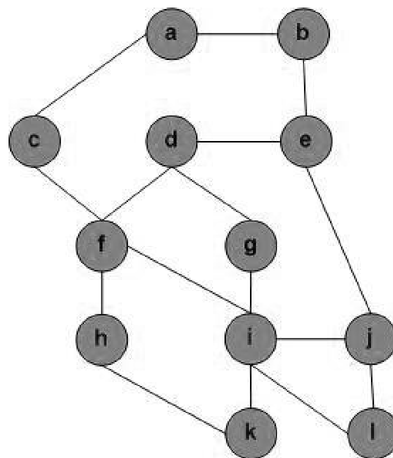


Fig. 5.8 : Un graphe non orienté pour illustrer les parcours

5.3.1 Parcours générique

Considérons l'algorithme de parcours ci-dessous :

Soit L la liste des sommets représentant le chemin de parcours $L \leftarrow s$ /* s est le sommet de départ */ **Pour** $i \leftarrow 1$ à $n-1$ **Faire** Mettre $\mathcal{B}(L)$ à l'ensemble des sommets de $S - L$ adjacents à L /* Bordure de L */ Choisir un sommet v dans $\mathcal{B}(L)$ $L \leftarrow L.(v)$ /* insérer v à la fin de L */ **FinPour**

A chaque étape, cet algorithme choisit (de façon aléatoire) un sommet parmi les sommets qui sont adjacents aux sommets déjà visités (ceux de L) et qui ne sont pas encore visités (i.e., qui appartiennent à l'ensemble $S - L$).

Pour le graphe de Fig. 4.5, cet algorithme générique s'applique comme suit :

Supposons que le sommet a est sélectionné pour départ, alors $L = \langle a \rangle$
Nous donnons ci-après l'état de la liste L après chaque itération de la boucle Pour :

Itération 1 :
$\mathcal{B}(L) = \langle b, c \rangle$
Supposons que b est choisi
$L = \langle a, b \rangle$

Itération 2 :
$\mathcal{B}(L) = \langle c, e \rangle$
Supposons que e est choisi
$L = \langle a, b, e \rangle$

Itération 3 :
$\mathcal{B}(L) = \langle c, d, j \rangle$
Supposons que d est choisi
$L = \langle a, b, e, d \rangle$

Itération 4 :
$\mathcal{B}(L) = \langle c, f, g, j \rangle$
Supposons que c est choisi
$L = \langle a, b, e, d, c \rangle$

Itération 5 :
$\mathcal{B}(L) = \langle f, g, j \rangle$
Supposons que g est choisi
$L = \langle a, b, e, d, c, g \rangle$

Itération 6 :

$\mathcal{B}(L) = \langle f, i, j \rangle$

Supposons que f est choisi

$L = \langle a, b, e, d, c, g, f \rangle$

Itération 7 :

$\mathcal{B}(L) = \langle h, i, j \rangle$

Supposons que j est choisi

$L = \langle a, b, e, d, c, g, f, j \rangle$

Itération 8 :

$\mathcal{B}(L) = \langle h, i, l \rangle$

Supposons que h est choisi

$L = \langle a, b, e, d, c, g, f, j, h \rangle$

Itération 9 :

$\mathcal{B}(L) = \langle i, l, k \rangle$

Supposons que i est choisi

$L = \langle a, b, e, d, c, g, f, j, h, i \rangle$

Itération 10 :

$\mathcal{B}(L) = \langle l, k \rangle$

Supposons que k est choisi

$L = \langle a, b, e, d, c, g, f, j, h, i, k \rangle$

Itération 11 :

$\mathcal{B}(L) = \langle l \rangle$

l est le seul choix

$L = \langle a, b, e, d, c, g, f, j, h, i, k, l \rangle$

5.3.2 Parcours en largeur

Pour décrire le parcours en largeur, on utilise une notion de distance entre les sommets :

- Un sommet est à distance 1 du sommet de départ s s'il est parmi les voisins de s, $voisins(s)$;
- Il est à distance 2 de s s'il est parmi les voisins des sommets qui sont à distance 1 à s, i.e., parmi $voisins(voisisn(s))$;
- Il est à distance 3 de s s'il est parmi les voisins des sommets qui sont à distance 2 à s, i.e., il y a un chemin court entre ce sommet et s passant par un sommet à distance 1 et un sommet à distance 2 …

L'algorithme de parcours en largeur visite en premier temps tous les sommets à distance 1 du sommet départ s, puis tous les sommets à distance 2 de s, puis tous les sommets à distance 3 . . . (d'où son nom parcours en largeur).

Algorithme :

Soit s sommet de départ, $s \in S$

Pour ranger les sommets selon leur distance de s, on utilise une file F de sommets visités

Soit L la liste de parcours, $L \leftarrow \langle \quad \rangle$

1. $ENFILER(s, F)$
   $L \leftarrow L.(s)$ /* insérer s à la fin de L */
2. On visite les voisins de la tête de F, $voisins(TETE(F))$. On les enfile s'ils ne sont pas dans F, ni présents dans L (non visités).
   Pour chaque élément e de $voisins(TETE(F))$ **Faire**
   Si $e \notin L$ **ET** $e \notin F$ **Alors**
   $ENFILER(e, F)$
   $L \leftarrow L.(e)$ /* insérer e à la fin de L */
   FinSi
   FinPour
3. $DEFILER(F)$ (On supprime la tête de F).
4. On recommence au point 2 tant que F n'est pas vide.

En appliquant cet algorithme sur le graphe de Fig. 4.5, l'état de la liste de parcours L la file F seront comme suit :

- Point 1 : Supposons que le sommet a est choisi pour départ, $L = \langle a \rangle$; $F = [a]$;
- Boucle
  - Itération 1
    - Point 2 : $F = [a, b, c]$; $L = \langle a, b, c \rangle$;
    - Point 3 : $F = [b, c]$
  - Itération 2
    - Point 2 : $F = [b, c, e]$; $L = \langle a, b, c, e \rangle$;
    - Point 3 : $F = [c, e]$
  - Itération 3
    - Point 2 : $F = [c, e, f]$; $L = \langle a, b, c, e, f \rangle$;
    - Point 3 : $F = [e, f]$
  - Itération 4
    - Point 2 : $F = [e, f, d, j]$; $L = \langle a, b, c, e, f, d, j \rangle$
    - Point 3 : $F = [f, d, j]$
  - Itération 5
    - Point 2 : $F = [f, d, j, h, i]$; $L = \langle a, b, c, e, f, d, j, h, i \rangle$
    - Point 3 : $F = [d, j, h, i]$
  - Itération 6
    - Point 2 : $F = [d, j, h, i, g]$; $L = \langle a, b, c, e, f, d, j, h, i, g \rangle$
    - Point 3 : $F = [j, h, i, g]$
  - Itération 7
    - Point 2 : $F = [j, h, i, g, l]$; $L = \langle a, b, c, e, f, d, j, h, i, g, l \rangle$;
    - Point 3 : $F = [h, i, g, l]$
  - Itération 8
    - Point 2 : $F = [h, i, g, l, k]$; $L = \langle a, b, c, e, f, d, j, h, i, g, l, k \rangle$;
    - Point 3 : $F = [i, g, l, k]$
  - Itération 9
    - Point 2 : $F = [h, i, g, l, k]$; $L = \langle a, b, c, e, f, d, j, h, i, g, l, k \rangle$;
    - Point 3 : $F = [g, l, k]$
  - Itération 10
    - Point 2 : $F = [g, l, k]$; $L = \langle a, b, c, e, f, d, j, h, i, g, l, k \rangle$;
    - Point 3 : $F = [l, k]$
  - Itération 11
    - Point 2 : $F = [l, k]$; $L = \langle a, b, c, e, f, d, j, h, i, g, l, k \rangle$;
    - Point 3 : $F = [k]$
  - Itération 12
    - Point 2 : $F = [k]$; $L = \langle a, b, c, e, f, d, j, h, i, g, l, k \rangle$;
    - Point 3 : $F = [\ \ ]$

5.3.3 Parcours en profondeur

Algorithme :

Soit s sommet de départ, $s \in S$

Dans le parcours en profondeur, on utilise une pile P de sommets visités

Soit L la liste de parcours, $L \leftarrow \langle\ \ \rangle$

1. $EMPILER(s, P)$

$L \leftarrow L.(s)$ /* insérer s à la fin de L */

2. **Si** le sommet de P, $SOMMET(P)$, a des voisins qui ne sont pas dans P, ni présents dans L (non visités) **Alors**

   On sélectionne un de ces voisins, soit e, et on l'empile dans $P, EMPILER(e, P)$; On marque e comme visité (découvert), $L \leftarrow L.(e)$

   Sinon

   $DEPILER(P)$ /* on supprime l'élément du sommet de la pile */

3. On recommence au point 2 tant que P n'est pas vide.

Le parcours en profondeur traite toujours en priorité les sommets de graphe les plus tard découvertes.

En appliquant cet algorithme sur le graphe de Fig. 4.5, l'état de la liste de parcours L la pile P seront comme suit :

- Point 1 : Supposons que le sommet a est choisi pour départ, $L = \langle a \rangle$; $P = [a]$;
- Boucle
  - Itération 1
    - Sommet de P a a deux voisins non visités et qui ne sont pas dans P, qui sont $\{b, c\}$, supposons que b est sélectionné, alors $P = \begin{bmatrix} b \\ a \end{bmatrix}$; $L = \langle a, b \rangle$
  - Itération 2
    - Sommet de P b a un voisin non visité et qui n'est pas dans P, qui est $\{e\}$, alors $P = \begin{bmatrix} e \\ b \\ a \end{bmatrix}$; $L = \langle a, b, e \rangle$
  - Itération 3
    - Sommet de P e a deux voisins non visités et qui ne sont pas dans P, qui sont $\{d, j\}$, supposons que d est sélectionné, alors $P = \begin{bmatrix} d \\ e \\ b \\ a \end{bmatrix}$; $L = \langle a, b, e, d \rangle$
  - Itération 4
    - Sommet de P d a deux voisins non visités et qui ne sont pas dans P, qui sont $\{f, g\}$, supposons que g est sélectionné, alors $P = \begin{bmatrix} g \\ d \\ e \\ b \\ a \end{bmatrix}$; $L = \langle a, b, e, d, g \rangle$
  - Itération 5
    - Sommet de P g a un voisin non visité et qui n'est pas dans P, qui est $\{i\}$, alors $P = \begin{bmatrix} i \\ g \\ d \\ e \\ b \\ a \end{bmatrix}$; $L = \langle a, b, e, d, g, i \rangle$
  - Itération 6
    - Sommet de P i a trois voisins non visités et qui ne sont pas dans P, qui sont $\{j, l, k\}$, supposons que l est sélectionné, alors $P = \begin{bmatrix} l \\ i \\ g \\ d \\ e \\ b \\ a \end{bmatrix}$; $L = \langle a, b, e, d, g, i, l \rangle$

- Itération 7
  - Sommet de P l a un voisin non visité et qui n'est pas dans P, qui est $\{j\}$, , alors $P = \begin{bmatrix} j \\ l \\ i \\ g \\ d \\ e \\ b \\ a \end{bmatrix}$; $L = \langle a, b, e, d, g, i, l, j \rangle$
- Itération 8
  - Sommet de P j n'a aucun voisin non visité et qui n'est pas dans P, alors $P = \begin{bmatrix} l \\ i \\ g \\ d \\ e \\ b \\ a \end{bmatrix}$;

    $L = \langle a, b, e, d, g, i, l, j \rangle$
  - Sommet de P l n'a aucun voisin non visité et qui n'est pas dans P, alors $P = \begin{bmatrix} i \\ g \\ d \\ e \\ b \\ a \end{bmatrix}$;

    $L = \langle a, b, e, d, g, i, l, j \rangle$
- Itération 9
  - Sommet de P i a un voisin non visité et qui n'est pas dans P, qui est $\{k\}$, alors $P = \begin{bmatrix} k \\ i \\ g \\ d \\ e \\ b \\ a \end{bmatrix}$; $L = \langle a, b, e, d, g, i, l, j, k \rangle$
- Itération 10
  - Sommet de P k a un voisin non visité et qui n'est pas dans P, qui est $\{h\}$, alors $P = \begin{bmatrix} h \\ k \\ i \\ g \\ d \\ e \\ b \\ a \end{bmatrix}$; $L = \langle a, b, e, d, g, i, l, j, k, h \rangle$
- Itération 10

- Sommet de P h a un voisin non visité et qui n'est pas dans P, qui est $\{f\}$, alors $P = \begin{bmatrix} f \\ h \\ k \\ i \\ g \\ d \\ e \\ b \\ a \end{bmatrix}$; $L = \langle a, b, e, d, g, i, l, j, k, h, f \rangle$

- Itération 10
  - Sommet de P f a un voisin non visité et qui n'est pas dans P, qui est $\{c\}$, alors $P = \begin{bmatrix} c \\ f \\ h \\ k \\ i \\ g \\ d \\ e \\ b \\ a \end{bmatrix}$; $L = \langle a, b, e, d, g, i, l, j, k, h, f, c \rangle$

- Itération 11 - 19
  - Tout Sommet de P n'a aucun voisin non visité et qui n'est pas dans P, alors on dépile P jusqu'à il est vide ; $L = \langle a, b, e, d, g, i, l, j, k, h, f, c \rangle$

Références (Livres et polycopiés)

- Beauquier, D., Berstel, J., & Chrétienne, P. (1992). Éléments d'algorithmique (p. 463pp). Masson.
- Perifel, S. (2014). Complexité algorithmique. Ellipses.
- Wilf, H. S. (2002). Algorithms and complexity. AK Peters/CRC Press.
- Mark Allen Weiss, Data Structures and Algorithm Analysis in Java, Pearson, Third Edition, 2012.
- William J. Collins, Data Structures and the Java Collections Framework, Wiley, 2011.
- Robert Sedgewick. « Algorithmes en langage C », InterEditions, 1991.
- Ouvrage collectif (coordination Luc Albert), « Cours et exercices d'informatique », Vuibert.
- Donald E. Knuth, The Art of Computer Programming, Volume 1, 2 et 3, Addison-Wesley.
- Laurent Signac, Algorithmique et Programmation, Ecole Supérieure d'Ingénieurs de Poitiers.