CSC_3IN03_TA (ex IN103)
Résolution de problèmes algorithmiques
une approche fondée sur les structures de données

Alexandre Chapoutot

Année académique 2024-2025

https://perso.ensta-paris.fr/~chapoutot/teaching/in103/

Avant de commencer

Moi == (encore) Alexandre Chapoutot
U2IS, bureau R217

- Ne pas hésiter à poser des questions, même en amphi.
- Posez votre question que vous jugez «bête» : d'autres ici ont la même.
- Ne pas hésiter à dire *M'sieur, vous vous z'êtes pas trompé sur xyz ?*
- Mon bureau et mon mail sont facilement *accessibles*.
- Terminez et conservez vos exercices, vos programmes.

Remise en contexte

Pour le moment vous avez étudié, par ordre chronologique

- la programmation en langage C (maîtriser la syntaxe de C)
- les bases de l'algorithmique (étudier et mettre en œuvre une méthodologie rigoureuse de conception des algorithmes)

Dans ce cours, nous poursuivons l'apprentissage de la pensée algorithmique :

- en étudiant des algorithmes connus ;
- en se fondant sur une organisation des données adaptées => structures de données.

Pour cela nous utiliserons :

- les traits caractéristiques du langage C, notamment les pointeurs ;
- une bibliothèques de structures de données conçue spécialement, nommée libin103.

Pourquoi une bibliothèque spécifique au cours ?

- Par ce que le langage C n'est pas doté d'une bibliothèque de structure de données comme d'autres langages (p. ex., C++, OCaml, Java, ...)
- Il existe des bibliothèques de structures de données en C
  - GLib (associée à GNOME)
  - Gnulib
  - SGLIB
  - liblfds
  - et plein de trucs sur GitHub

  Mais elles ne sont pas faciles à prendre en main ou incomplètes.
- La bibliothèque libin103 a été conçue pour simplifier son utilisation et illustrer tous les concepts du cours, pas pour les performances

Remarque : L'utilisation d'une bibliothèque en IN103 vous facilitera la prise en main de bibliothèques dans d'autres langages, en particulier, le langage C++ étudié en 2A (maths/info/méca).

Déroulement du cours et évaluation

Sauf exception, les séances sont composées de la manière suivante

- 1h de cours magistral (avec beaucoup d'informations)
- 2h de TD/TP (avec encore beaucoup d'informations complémentaires)

et $x \gg 0$ secondes de travail à la maison

La note sera composée

- Notes de QCM hebodmadaires (QCM) ;
- Note de l'examen intermédiaire (E_1) ;
- Note de l'examen final (E_2).

en suivant la règle, pour les personnes sans aménagement :

$$\text{note} = 0.1 \times \text{QCM} + 0.25 \times E_1 + 0.65 \times E_2$$

ou en suivant la règle, pour les personnes ayant un aménagement :

$$\text{note} = 1.25 \times (0.1 \times \text{QCM} + 0.25 \times E_1 + 0.65 \times E_2)$$

Quelques conseils pour réussir

Mantra de l'apprentissage en IN103 (mais pas que)

- Lire n'est pas comprendre
- Copier-coller n'est pas comprendre
- Écrire est comprendre

Méthode de travail (à appliquer régulièrement)

- Réécrivez/Compilez/Exécutez les corrections des exercices ;
- Questionnez vous sur les lignes qui sont utlisées pour réaliser le programme (pourquoi comme ça ?, est-ce qu'il y a une autre façon de faire ?, ...) ;
- Essayez d'expliquer à haute voix les programmes que vous (ré)écrivez au canard en plastique, cf MO101 ;-) ou à un ou une camarade.

La clef de la réussite : s'entraîner, s'entraîner, s'entraîner, ...

Agenda

- 4 février 2025 : Quelques rappels sur le langage C et un peu plus
  - Compilation séparée, make, Pointeurs
- 11 février 2025 : Structures de données linéaires
  - Listes chaînées, Piles, Files, Ensembles
- 18 février 2025 : Structures arborescentes – partie 1
  - Arbres binaires, Arbres binaires de recherche
- 4 mars 2025 : Examen intermediaire (3h)
- 11 mars 2025 : Structures arborescentes – partie 2
  - Tas, Ensembles disjoints (union-find)
- 18 mars 2025 : Graphes – partie 1
  - Représentations, Parcours de graphes (largeur/profondeur, tri topologique)
- 25 mars 2025 : Graphes – partie 2
  - Composantes fortement connexes, Plus court chemin, Arbre couvrant de poids minimum
- 1 avril 2025 : Examen (3h)

Remarque : Le contenu des séances pourra être adapté en fonction des envies de l'enseignant

Plan du cours

1. Compilation séparée
2. Les Makefiles
3. Retour sur les pointeurs
4. La bibliothèque libin103
5. Listes chaînées
6. Files
7. Piles
8. Ensembles
9. Arbres
10. Arbres binaires
11. Parcours d'arbres
12. Arbres binaires de recherche
13. Arbres binaires de recherche équilibrés
14. Tas
15. Ensembles disjoints
16. Graphes non orientés
17. Graphes orientés
18. Représentations mathématico-informatiques des (di)graphes
19. Parcours de graphes
20. Connectivité des (di)graphes
    - Composantes connexes
    - Composantes fortement connexes
21. (di)graphes pondérés
22. Arbre couvrant de poids minimum
23. Plus court chemin

Compilation séparée

Rappels sur le langage C – 1

Principales constructions du langage :

- `<instruction>;`
  Exemples : affectation `x = y + 3;` ou appel de fonction `f(x);`
- `{ <instructions> }` – bloc d'instructions
  Exemple : `{ int x = 2; int y = x + 3; }`
- `if (<condition>) <instruction(s)> else <instruction(s)>`
  Exemple :
  ```
  if (x % 2 == 0) { printf ("pair\n"); } else { printf ("impair\n"); }
  ```
- `while (<condition>) <instruction(s)>`
  Exemple : `int x = 10; while (x > 0) { printf ("%d\n", x); x--;}`
- `for (<init>; <condition>; <step>) <instruction(s)>`
  Exemple : `for (int x = 0; x<10; x++) { printf ("%d\n", x); }`
- `switch (<exp>) { case: <instruction(s)>; ... defaut: <instruction(s)>; }`
  Exemple :
  ```
  switch (c) { case 'q': return 0; default: printf ("joue␣encore\n"); }
  ```

Rappels sur le langage C – 2

- Types de données de base

Nom	Taille	Description
`char`	1 byte	caractère dans l'encodage ASCII, `'a'`, `'\n'`
`int`	4 bytes	valeur entière signée, `97`, `0xFF`, `0777`
`long long int`	8 bytes	valeur entière signée de grande taille
`float`	4 bytes	valeur flottante simple précision
`double`	8 bytes	valeur flottante double précision

- Les principales opérations

Symboles	Description
`+, -, *, /, %`	Opérations arithmétiques
`==, <=, >=, !=`	Opérations de comparaison
`&&, ||, !`	Opérations booléennes
`?:`	Expression conditionnelle (`x >= 0 ? "POS": "NEG"`)
`i++, i--`	Post incrémentation/décrémentation

Les types

Information essentielle pour une variable : donne l'espace mémoire nécessaire pour la stocker (`sizeof`) et les opérations admissibles pour la manipuler.

Rappels sur le langage C – 3

- Types de données composites

Nom	Description
`struct`	Structure regroupant plusieurs variables de types potentiellement différents dans un même type. (`struct point {double x; double y};`) et accès avec la notation pointée (`struct point p; p.x = 2.0;`)
`enum`	Structure regroupant plusieurs constantes entières dans un même type. (`enum color {BLACK, WHITE};`) et (`enum color col = WHITE;`)
`type a[SIZE]`	Tableau statique de SIZE éléments du même type `type`. (`int a[3] = {1, 2, 3};`) et (`a[0]`) est le premier élément et (`a[2]`) est le dernier élément.

- Les chaînes de caractères
  Ce sont des tableaux de caractères terminés par le caractère `'\0'`.

Rappels sur le langage C – 4

Les pointeurs sont déclarés grâce à la présence du caractère `*` à droite du type lors de la déclaration de variable

- `int *a;` *a* est un pointeur sur un entier ;
- `char *b;` *b* est un pointeur sur un caractère ;
- `int *c[2];` *c* est un tableau de pointeurs sur des entiers ;
- `struct point *p;` *p* est un pointeur sur une structure de type `point`.

Remarque : *Pointeurs = Adresses mémoires*

Manipulation des pointeurs :

- `a` représente l'adresse de l'entier pointé ;
- `*a` représente la valeur stockée à l'adresse (déréferencement) ;
- `int x; a = &x;` & donne l'adresse d'une variable ;
- `p->x = 2.0;` identique à `(*p).x`, raccourci pour déréfencer un pointeur sur une structure.

Rappels sur le langage C – 5

Tableaux

- Allocation statique
  - si la taille est connue à la compilation et de taille modeste ;
  - si le tableau n'est pas valeur de retour d'une fonction.

```
#define SIZE 20
char name[SIZE];
```

- Allocation dynamique dans les autres situations
  - Allocation mémoire : `malloc`
  - Libération mémoire : `free`

```
double *array = malloc (10 * sizeof(double));
if (array == NULL) { /* Erreur \‘a traiter */ }
```

Rappels sur le langage C – 6

Une fonction est décrite par

- un prototype (ou signature) donnant le type de la valeur retournée, son nom et le type de ses paramètres ;
- un bloc d'instructions.

```c
double sqr (double x) {
  return x * x;
}
```

- Prototype : `double sqr (double)`
- Bloc d'instructions :

  `{ return x * x; }`

Fonction principale

Un programme écrit en langage C a toujours une unique fonction principale nommée `main` dont le prototype complet est `int main (int, char**)`.

Bibliothèque standard de C

Ensemble de fonctions organisées par catégories dont

- Gestion de la mémoire et utilitaires (`#include <stdlib.h>`)
  Fonctions : `malloc`, `free`, `atoi`, `rand`, ...
- Entrées/Sorties (`#include <stdio.h>`)
  Fonctions : `printf`, `scanf`, `fopen`, `fclose`, `fgets`, `fputs`, `perror`, ...
- Chaînes de caractères (`#include <string.h>`)
  Fonctions : `strlen`, `strncpy`, `strncmp`, ...
- Mathématiques (`#include <math.h>`)
  Fonctions : `sin`, `cos`, `pow`, `ceil`, `floor`, ...
- Classification des caractères (`#include <ctype.h>`)
  Fonctions : `isalpha`, `isupper`, `isspace`, `isdigit`, ...
- Valeurs booléennes (`#include <stdbool.h>`)
  Constantes : `true`, `false`

Anatomie d'un programme C – 1

```c
#include <stdio.h>

/* Defintions des fonctions auxiliaires */
double sqr (double x) {
  return x * x;
}

int main () {
  double x = 2.0;
  printf ("Sqr(%f) = %f\n", x, sqr(x));
  return 0;
}
```

- Une fonction doit être connue avant d'être utilisée dans la fonction principale.

Compilation / Exécution

- Compilation : `gcc -Wall -Werror sqr.c -o sqr.x`
- Exécution : `./sqr.x`

Anatomie d'un programme C – 2

```c
#include <stdio.h>

/* Declaration des fonctions auxiliaires */
double sqr (double x);  /* Notez ; terminal et l'absence de corps */

int main () {
  double x = 2.0;
  printf ("Sqr(%f) = %f\n", x, sqr(x));
  return 0;
}

/* Defintions des fonctions auxiliaires */
double sqr (double x) {
  return x * x;
}
```

- Une autre façon de faire est de déclarer les fonctions, c'est-à-dire, donner leurs prototypes ;
- Puis définir les fonctions, c'est-à-dire donner leur code.

Compilation séparée

Dans le cas de projets de programmation avec de nombreuses fonctions,

- il est commode de ne pas définir toutes les fonctions dans le fichier contenant la fonction principale.
  Conséquences : travail collaboratif et maintenance du code simplifiés
- il est également plus facile de partager du code entre différents projets si les fonctions ne sont pas toutes dans le même fichier.
  Conséquences : facilite la réutilisation

Exemple

```c
/* Fichier sqr.h */
double sqr (double x);

/* Fichier sqr.c */
double sqr (double x) {
  return x * x;
}
```

```c
/* Fichier main_sqr.c */
#include <stdio.h>
#include "sqr.h"

int main () {
  double x = 2.0;
  printf ("Sqr(%f) = %f\n", x, sqr(x));
  return 0;
}
```

Compilation séparée

Plusieurs fichiers :

- Les fichiers .c contenant les définitions des fonctions ;
- Les fichiers .h contenant les déclarations des fonctions qui sont exportées (mise à disposition) par le fichier .c ;

Convention: Sauf pour le fichier contenant la fonction principale, nous aurons des couples de fichiers .h/.c (Ex., sqr.h / sqr.c)

Compilation

Pour l'exemple précédent, on peut compiler les fichiers un à un

- `gcc -Wall -Werror -c sqr.c`
- `gcc -Wall -Werror -c main-sqr.c -o main-sqr.o`
- `gcc -Wall -Werror main-sqr.o sqr.o -o main-sqr.x`

ou en forme condensée

- `gcc -Wall -Werror main-sqr.c sqr.c -o main-sqr.x`

Compilation séparée

Rappel : `#include` remplace la ligne par le contenu du fichier donné en argument.

- Si `#include "fichier.h"` cherche le fichier dans le répertoire courant puis dans les répertoires du système (Ex. `/usr/include` et `/usr/local/include`).
- Si `#include <fichier.h>` cherche le fichier les répertoires du système.

Résultat de la commande (appel au pré-processeur seul) : `gcc -E main-sqr.c`

```
# 1 "main-sqr.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 367 "<built-in>" 3
# 1 "<command␣line>" 1
# 1 "<built-in>" 2
# 1 "main-sqr.c" 2
# 1 "./sqr.h" 1

double sqr (double x);
# 2 "main-sqr.c" 2

int main () {
```

Le préprocesseur supprime tous les commentaires, fait les inclusions et autres traitements.

Les lignes de sorties ont la forme
`# numeroligne fichier drapeau`
avec comme valeurs principales du drapeau : 1 début de fichier, 2 retour au fichier, 3 le texte vient d'un fichier en-tête.

Note : `#include <stdio.h>` a été ignorée.

Compilation séparée et bibliothèques logicielles

Pour faciliter la réutilisation de fonctions, au lieu de dsitribuer le code source (fichiers .c), une bibliothèque logicielle (collection de .o) peut être distribuée :

- Bibliothèque statique, par exemple `libin103.a`,
  bibliothèque concaténée au fichier exécutable lors de la compilation.
- Bibliothèque dynamique, par exemple `libin103.so`,
  bibliothèque est chargée à l'exécution du programme.

Remarque : Un ou plusieurs fichiers .h peuvent être associés à la bibliothèque pour exposer les fonctions accessibles.

Exemple pour la bibliothèque libin103

```
bitree.h bitreealg.h bistree.h graph.h list.h stack.h
heap.h queue.h uf.h set.h
```

Exemple d'utilisation de bibliothèque logicielle

Exemple : arborescence d'un projet (avant compilation)

```
Makefile myinclude/ lib1/ lib2/ main.c

./myinclude/: bonjour.h

./lib1/: Makefile bonjour1.c

./lib2/: Makefile bonjour2.c
```

Fichier `main.c`

```c
#include <stdlib.h>

#include "bonjour.h"

int main (int argc, char** argv) {

  bonjour("Alex");

  return EXIT_SUCCESS;
}
```

Fichier `include/bonjour.h`

```c
void bonjour (char* str);
```

Exemple d'utilisation de bibliothèque logicielle

Exemple : arborescence d'un projet (avant compilation)

```
Makefile myinclude/ lib1/ lib2/ main.c

./myinclude/: bonjour.h

./lib1/: Makefile bonjour1.c

./lib2/: Makefile bonjour2.c
```

Fichier `lib1/bonjour1.c`

```c
#include <stdio.h>

void bonjour (char* name) {
  printf ("Salut␣%s\n", name);
}
```

Fichier `lib2/bonjour2.c`

```c
#include <stdio.h>

void bonjour (char* name) {
  printf ("Hello␣%s\n", name);
}
```

Exemple d'utilisation de bibliothèque logicielle

- Règle (simplifiée) de compilation pour `main-static1.x`

```
gcc -Iinclude -o main-static1.x main.c ./lib1/libbonjour.a
```

- Règle (simplifiée) de compilation pour `main-dynamic.x`

```
gcc -Iinclude -o main-dynamic.x main.c -Llib2 -lbonjour
```

Nouvelles options de `gcc`

- `-I` définit un répertoire dans lequel trouver des fichiers .h
- `-L` définit un répertoire dans lequel trouver des bibliothèques .a ou .so
- `-lbonjour` drapeau indiquant l'utilisation de la bibliothèques `libbonjour.{a,so}`
  Note : par défaut, `gcc` lie les bibliothèques dynamiques.

Exemple d'utilisation de bibliothèque logicielle

Exemple : arborescence d'un projet (après compilation)

```
Makefile myinclude/ lib1/ lib2/ main-dynamic.x main-static1.x
  main-static2.x main.c

myinclude/: bonjour.h

lib1/: Makefile bonjour1.c bonjour1.o libbonjour.a libbonjour.so

lib2/: Makefile bonjour2.c bonjour2.o libbonjour.a libbonjour.so
```

```
% ./main-static1.x
Salut Alex
% ./main-static2.x
Hello Alex
% ./code/librairies/main-dynamic.x
ld: Library not loaded: libbonjour.so
  Referenced from: ./main-dynamic.x
  Reason: image not found
% LD_LIBRARY_PATH=./lib1 ./main-dynamic.x
Salut Alex
% LD_LIBRARY_PATH=./lib2 ./main-dynamic.x
Hello Alex
```

Les Makefiles

Makefile

Constatation

Les lignes de compilation peuvent être longues et complexes. La compilation dans ce cas compile tous les fichiers concernés à chaque fois.

Exemple

```
gcc -Wall -Werror -I../include -L../lib -o my_prog.x my_prog.c algo.c -lm -lin103
```

Les fichiers Makefile permettent

- de sauvegarder les règles de compilation ;
- de gérer les dépendances (cf plus loin) pour limiter l'effort de compilation.

Dans ce cours nous allons utiliser l'outil GNU Make mais d'autres outils existent (par exemple, CMake qui s'appuie sur GNU Make).

Anatomie d'un Makefile

Convention: Un fichier Makefile sera nommé `Makefile`.

Un Makefile contient

- des variables ;
- des règles (ou *recipes* dans la terminologie consacrée).

```
CC=gcc

hello.x: hello.c
	$(CC) -Wall -Werror -o hello.x hello.c
```

Makefile – variables

- Une variable n'est qu'une définition dans un Makefile
- Pour récupérer son contenu on utilise la notation `$()`.

Exemple – variable

Définition d'une variable `cc` donnant la commande pour compiler un fichier .c

```
CC=gcc
```

Exemple – utilisation de variable

Utlisation de la variable CC

```
$(CC) -Wall -Werror -o hello.x hello.c
```

Convention: les variables d'un Makefile sont placées en début de fichier avec des noms en majuscules.

Makefile – règles

Les règles de production ont la forme générale suivante

```
cible: dependance1 dependance2 ...
      actions
```

- `cible` : souvent le nom d'un fichier qui doit être le résultat des actions mais pas que.
- `dépendances` : éléments (séparés par un espace) qui sont nécessaires pour construire la cible
  - elles peuvent être des cibles (intermédiaires) qui doivent être créées avant l'exécutions des actions de la règle courante ;
  - elles peuvent être des noms de fichiers, ils doivent être présents.
- `actions` (recipes) : commandes `shell` pour construire la cible.

Point de vigilance

Vous DEVEZ utiliser une tabulation (touche ⇥) pour indenter vos actions ! !

Makefile – exécution

Pour exécuter les règles d'un Makefile

- Il faut utiliser la commande `make` dans le répertoire contenant le fichier `Makefile` ;
- Par défaut, la commande `make` (sans argument) exécute la première règle dans le fichier `Makefile` ;
- Pour exécuter une règle spécifique, il faut utiliser la commande

```
make target
```

Fonctionnement :

- Si une dépendance doit être reconstruite (issue d'une règle) ou avec une date de modification plus récente que la cible (pour les fichiers), la règle doit être exécutée.
- Une cible qui n'est pas un fichier rend la règle *PHONY*. Cette règle est exécutée inconditionnellement à chaque appel.

Makefile – autre exemple

Un ficher Makefile permet de sauvegarder les règles de compilation mais pas que :

Exemple plus complexe

```
CC=gcc

hello.x: hello.c
	$(CC) -Wall -Werror -o hello.x hello.c

clean: # PHONY recipe
	rm -f hello.o hello.x

distrib: $(USER)-livraison.tgz # PHONY recipe

$(USER)-livraison.tgz: hello.c
	mkdir $(USER)-livraison
	cp hello.c $(USER)-livraison
	tar czf $(USER)-livraison.tgz $(USER)-livraison
	rm -rf $(USER)-livraison

.PHONY: distrib clean # To avoid conflict with rep. with these names
```

Retour sur les pointeurs

Alias de type

Rappel

La commande `typedef` permet de créer des alias de types. Elle a la forme suivante

```
typedef <type existant> <nom>
```

- `typedef unsigned long int ulint;`
- `typedef struct point point_t;`
- `typedef double* vector_t;`
- Forme imbriquée (définition de type et création d'alias)

  ```
  typedef struct point_ { double x; double y; } point_t;
  ```

Point de vigilance

Cette commande ne définit pas un nouveau type mais juste un nouveau nom pour un type connu.

Transtypage (ou cast)

Transtypage

- Pour les types de bases : opération de conversion d'une valeur d'un type en une autre valeur d'un autre type.
- Pour les autres types : opération de changement de type d'une variable prise en compte à la compilation.

```
int x = (int) 3.14159; /* x = 3 */
```

Points de vigilance

- Possibilité de perte / corruption d'information ;
- La conversion entre entier et flottant change la représentation interne.
- Pour les pointeurs cela change uniquement la façon d'interpréter la zone mémoire associée.

Généricité en C

Un pointeur est une adresse correspondant à une zone mémoire dont la taille dépend du nombre et du type d'information qui y est stockée ⇒ pas possible d'affecter des pointeurs de type différents.

Le programme

```
#include <stdlib.h>

int main(int argc, char **argv)
{
  int* a = NULL;
  double *b = NULL;
  a = b;

  return 0;
}
```

génère une erreur à la compilation

```
toto.c:7:5: error: incompatible pointer types assigning to 'int␣*'from 'double␣*'
```

Généricité en C

- Dans différentes situations, il peut être utile de manipuler (stocker) des pointeurs sans connaître exactement les types de valeurs qu'ils référencent.
- Pour cela on utilise des pointeurs génériques `void*`

```c
int a = 10;
char b = 'c';

/* sans erreur */
void *p = &a;

/* sans erreur */
p = &b;
```

```c
#include <stdio.h>

int main()
{
    int a = 10;
    void *ptr = &a;
    printf("%d", *((int *)ptr));
    return 0;
}
```

Point de vigilance

Pour utiliser un pointeur générique, il faut le convertir dans le type de données adéquate $\Rightarrow$ il faut mémoriser (dans le code ou dans sa tête) ce qu'il représente.

Exemple

```
void swapg (void* a, void* b, int size) {
  void *tmp = malloc (size);
  /* void* memcpy(void *dst, void *src, size_t n); */
  memcpy (tmp, a, size); /* copie une zone memoire */
  memcpy (a, b, size);
  memcpy (b, tmp, size);
  free (tmp);
}

int main(int argc, char **argv)
{
  int a = 1; int b = 4;
  printf ("a = %d, b = %d\n", a, b);
  swapg((void*)&a, (void*)&b, sizeof(int)); /* param. int */
  printf ("a = %d, b = %d\n", a, b);

  double c = 1.1; double d = 4.1;
  printf ("c = %f, d = %f\n", c, d);
  swapg((void*)&c, (void*)&d, sizeof(double)); /* param. double */
  printf ("c = %f, d = %f\n", c, d);

  return 0;
}
```

Motivation pour une bibliothèque spécifique à IN103

- Par ce que le langage C n'est pas doté d'une bibliothèque de structure de données comme d'autres langages (p. ex., C++, OCaml, Java, ...)
- Il existe des bibliothèques de structures de données en C
  - GLib (associée à GNOME), Gnulib, SGLIB, liblfds
  - et plein de trucs sur GitHub

  Mais elles ne sont pas aisées à prendre en main ou incomplètes.
- La bibliothèque libin103 a été concue pour simplifier l'utilisation et illustrer tous les concepts du cours, pas pour les performances

Remarque

Beaucoup d'algorithmes sont efficaces car ils ont une gestion des données efficaces. Cette gestion s'appuie sur des structures de données, c'est-à-dire, une représentation des données pertinentes pour résoudre la classe de problèmes considérée.

La bibliothèque libin103

Tour d'horizon de la bibliothèque `libin103`

Pour les besoins de l'enseignement IN103, une bibliothèque logicielle a été développée, mettant en œuvre les principales structures de données et algorithmes utilisés durant le cours.

Les structures de données présentes sont :

- Listes chaînées (`list`), les Piles (`stack`), les Files (`queue`) et les Ensembles (`set`), en plusieurs versions :
  Pour les `int`, `double`, `char` et une version générique avec `void*`
- Les Arbres binaires (`bitree`), les Arbres binaires de recherche (`bistree`) et les Tas (`heap`), en plusieurs versions :
  Pour les `int`, `double`, `char` et une version générique avec `void*`
- Les Ensembles disjoints (`union-find`) uniquement en version pour les `int`
- Les Graphes (`graph`) en plusieurs versions :
  Pour les `int` et une version générique avec `void*`

A chaque séance du cours, une présentation du principe de fonctionnement des structures de données et des fonctions de manipulation seront réalisées.

Tour d'horizon de la bibliothèque `libin103`

Répertoire `libin103`

```
libin103-1.4/
|- include/
|- Makefile
|- Makefile.common
|- README.md
|- source/
|- templates/
|- tests/
```

Explications

- Dans le répertoire `include` : les fichiers entêtes `.h`, interface de programmation des structures de données ;
- Dans le répertoire `source` : les fichiers définitions `.c`. C'est également dans ce répertoire que sera stocké le fichier libin103.a ;
- Dans le répertoire `templates` : les fichiers qui servent à générer des versions spécialisées (`int`, `double`, etc.) des structures de données ;
- Dans le répertoire `tests` : les programmes qui ont servi à la mise au point de la bibliothèque. Ils peuvent également être lus comme des exemples d'utilisation de la bibliothèque.

Documentation HTML

`https://perso.ensta-paris.fr/~chapoutot/teaching/in103/refman/index.html`

Exemple : pour utiliser les listes chaînées

Fichiers .h

Dans le répertoire include, on trouvera les fichiers suivants

list.h, character_list.h, generic_list.h, integer_list.h, real_list.h

Fichier list.h

```
#ifndef LIST_H
#define LIST_H

#include "integer_list.h"
#include "real_list.h"
#include "character_list.h"
#include "generic_list.h"

#endif
```

Les autres fichiers seront présentés dans le reste du cours.

Listes chaînées

Motivation pour une nouvelle structure de données

Pour le moment, les tableaux ont été étudiés. Ils sont

- de taille fixe
- mis en œuvre par une zone mémoire contiguë d'éléments du même type
- alloués statiquement ou dynamiquement

Pensez aux exemples de M. Pessaux avec des fichiers dont la première ligne est le nombre de données à lire ensuite !

Les tableaux

- ne s'adaptent pas facilement à des données qui bougent beaucoup (insertion/suppression) ;
- offrent une représentation adaptée à l'adressage directe.

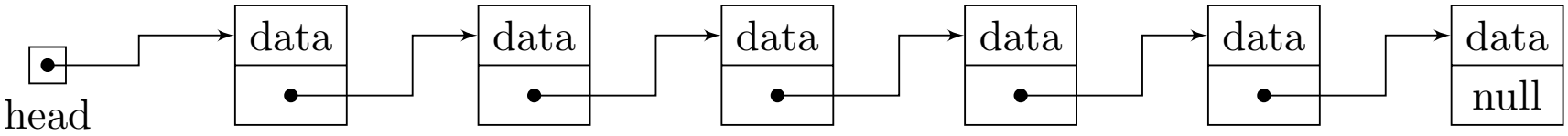
Implémentation des listes chaînées

Mathématiquement une `liste` peut être définie de manière inductive par :

- une liste vide
- un élément suivi d'une liste

$\Rightarrow$ type inductif de la forme : *list* ::= `Nil` | elem; *list*

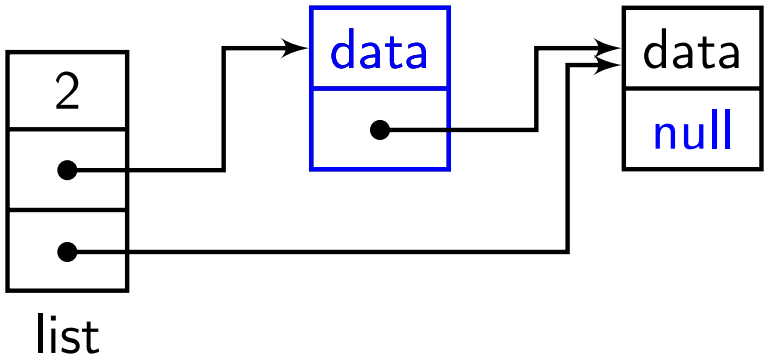
- Les langages fonctionnels (OCaml, Haskell, ...) permettent de définir des types inductifs mais pas le langage C.
- En langage C, les pointeurs sont utilisés pour créer un chaînage entre les éléments de la liste.



Zoologie des listes chaînées

- simplement/doublement chaînée ;
- circulaire, ...
- avec pointeur de fin, champs taille, etc.

Implémentation des listes chaînées

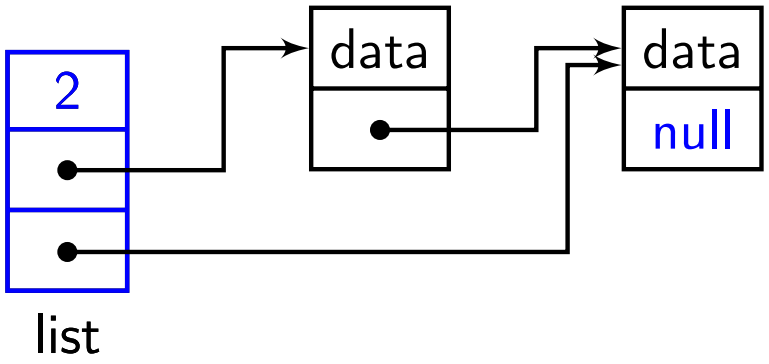


Une représentation des éléments de la liste

```
typedef struct integer_list_elmt_ {
  int data;  /* can be double or char or void*  */
  struct integer_list_elmt_t* next;
} integer_list_elmt_t;
```

Remarque : les listes permettent de gérer des données de manière dynamique (et les cellules de la chaîne sont répartis dans la mémoire) mais on perd l'accès direct aux éléments.

Implémentation des listes chaînées



Une représentation de la liste

```
typedef struct integer_list_ {
  int size;
  integer_list_elmt_t* head;
  integer_list_elmt_t* tail;
} integer_list_t;
```

Remarque : représentation choisie pour également faciliter la mise en œuvre d'autres structures de données (cf plus loin)

Interface de programmation [1]

Interface de programmation

Contient la liste des signatures des fonctions (ainsi qu'une courte description) permettant de manipuler une bibliothèque logicielle.

En général, l'API pour les structures de données contient

- les fonctions de créations/destructions de la structure de données
- les fonctions pour ajouter/supprimer des éléments dans la structure de données
- les fonctions permettant de récupérer de l'information (comme la taille)

1. En anglais, *Application Programming Interface* (API).

API des listes chaînées – 1

API des listes – Création/Destruction

```
void integer_list_init(integer_list_t*);

void integer_list_destroy(integer_list_t*);
```

API des listes – Accesseurs

```
int integer_list_size (integer_list_t*);

integer_list_elmt_t* integer_list_head(integer_list_t*);

integer_list_elmt_t* integer_list_tail(integer_list_t*);

integer_list_elmt_t* integer_list_next(integer_list_elmt_t*);

int integer_list_data(integer_list_elmt_t*);
```

API des listes chaînées – 2

API des listes – Prédicats

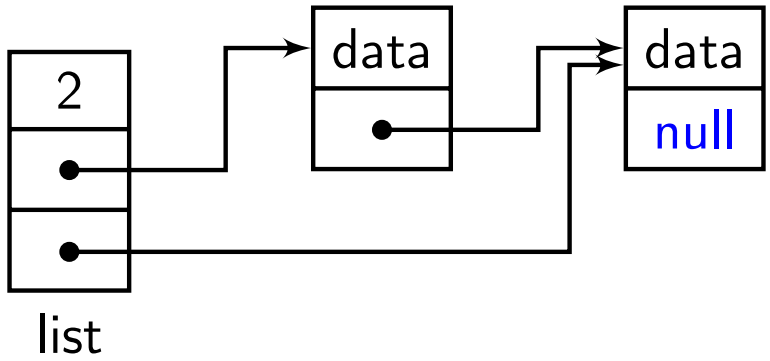
```
bool integer_list_is_head(integer_list_t*,
                          integer_list_elmt_t*);

bool integer_list_is_tail(integer_list_elmt_t*);
```

API des listes – Insertion/Suppression

```
int integer_list_ins_next(integer_list_t*,
                          integer_list_elmt_t*,
                          int data);

int integer_list_rem_next(integer_list_t*,
                          integer_list_elmt_t*,
                          int* data);
```

Insertion dans une liste chaînée

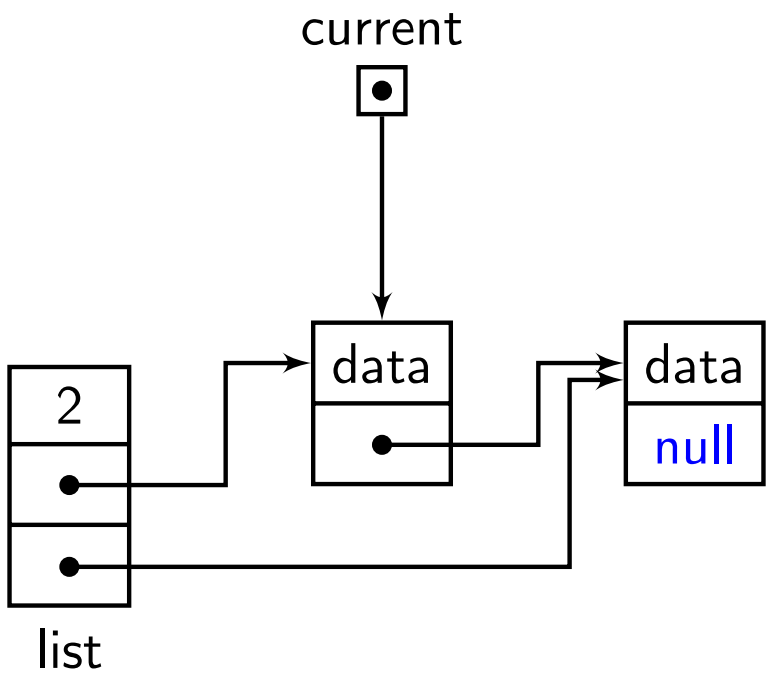
Suivant l'API, l'insertion dans une liste est réalisée à partir d'un élément.



Étapes d'insertion

Insertion dans une liste chaînée

Suivant l'API, l'insertion dans une liste est réalisée à partir d'un élément.

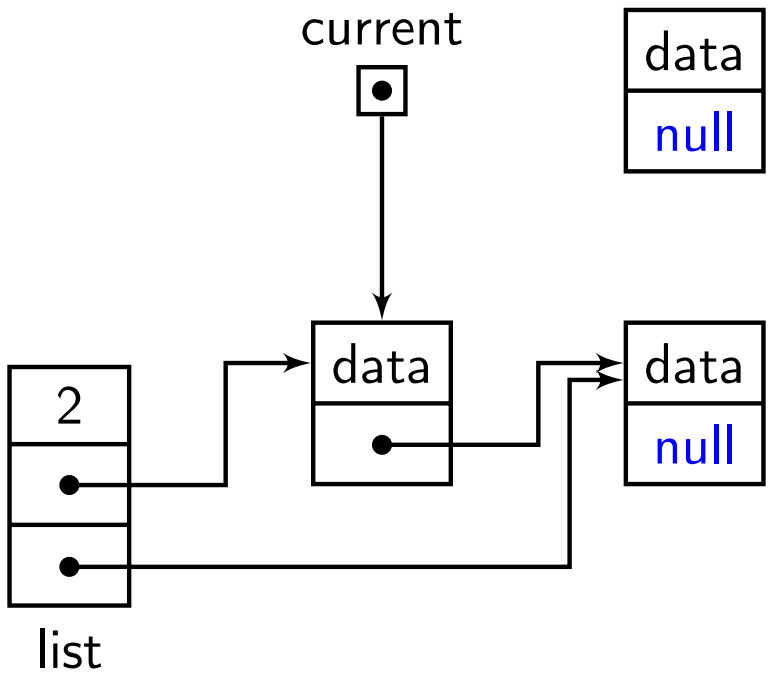


Étapes d'insertion

1. On donne l'élément après lequel insérer un nouvel élément

Insertion dans une liste chaînée

Suivant l'API, l'insertion dans une liste est réalisée à partir d'un élément.

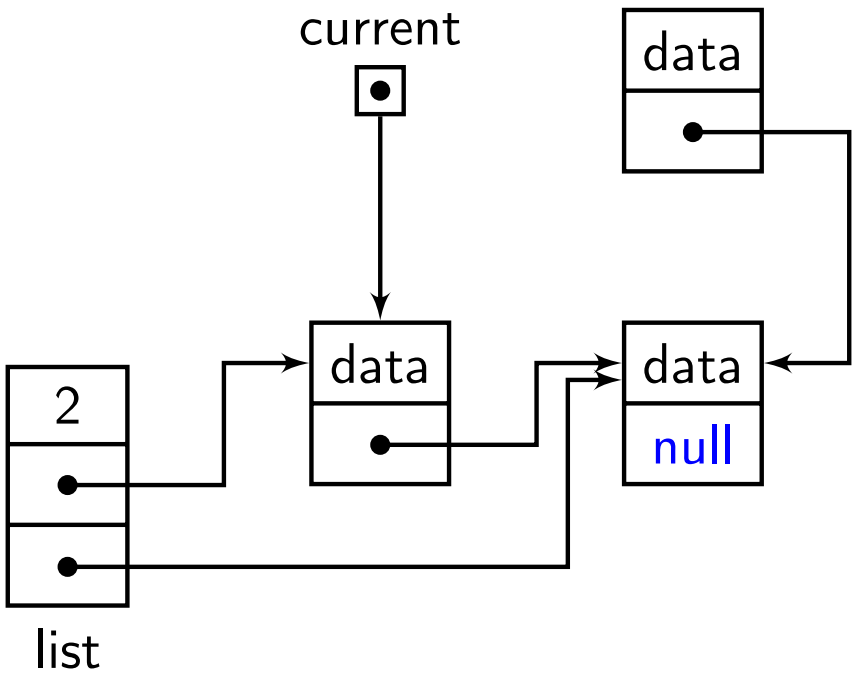


Étapes d'insertion

1. On donne l'élément après lequel insérer un nouvel élément
2. On crée une nouvelle cellule avec la nouvelle donnée

Insertion dans une liste chaînée

Suivant l'API, l'insertion dans une liste est réalisée à partir d'un élément.

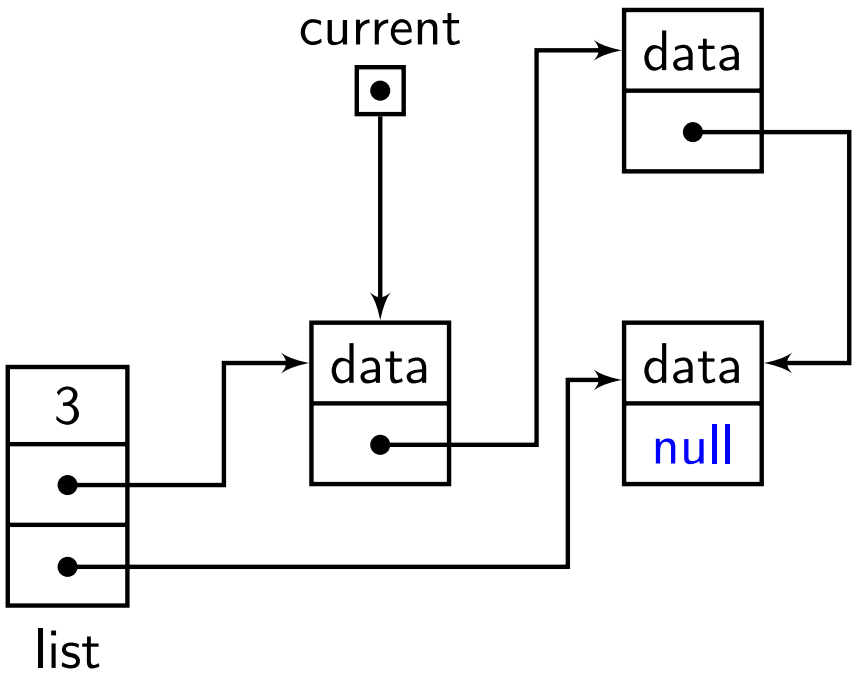


Étapes d'insertion

1. On donne l'élément après lequel insérer un nouvel élément
2. On crée une nouvelle cellule avec la nouvelle donnée
3. On fait le nouveau chaînage (attention à l'ordre des opérations)
4. On met à jour de la taille de la liste chaînée

Insertion dans une liste chaînée

Suivant l'API, l'insertion dans une liste est réalisée à partir d'un élément.



Étapes d'insertion

1. On donne l'élément après lequel insérer un nouvel élément
2. On crée une nouvelle cellule avec la nouvelle donnée
3. On fait le nouveau chaînage (attention à l'ordre des opérations)
4. On met à jour de la taille de la liste chaînée

Insertion dans une liste chaînée

Quelques comportements particuliers

- Que se passe-t-il si l'élément après lequel insérer est le pointeur `NULL` ?

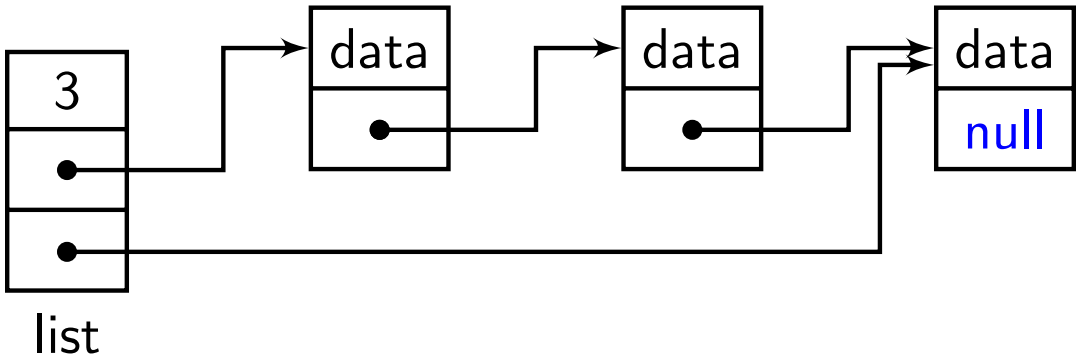
  Insertion en tête

- Comment faire une insertion en fin de liste ?

```
integer_list_ins_next(&list, integer_list_tail(&list), data);
```

Suppression dans une liste chaînée

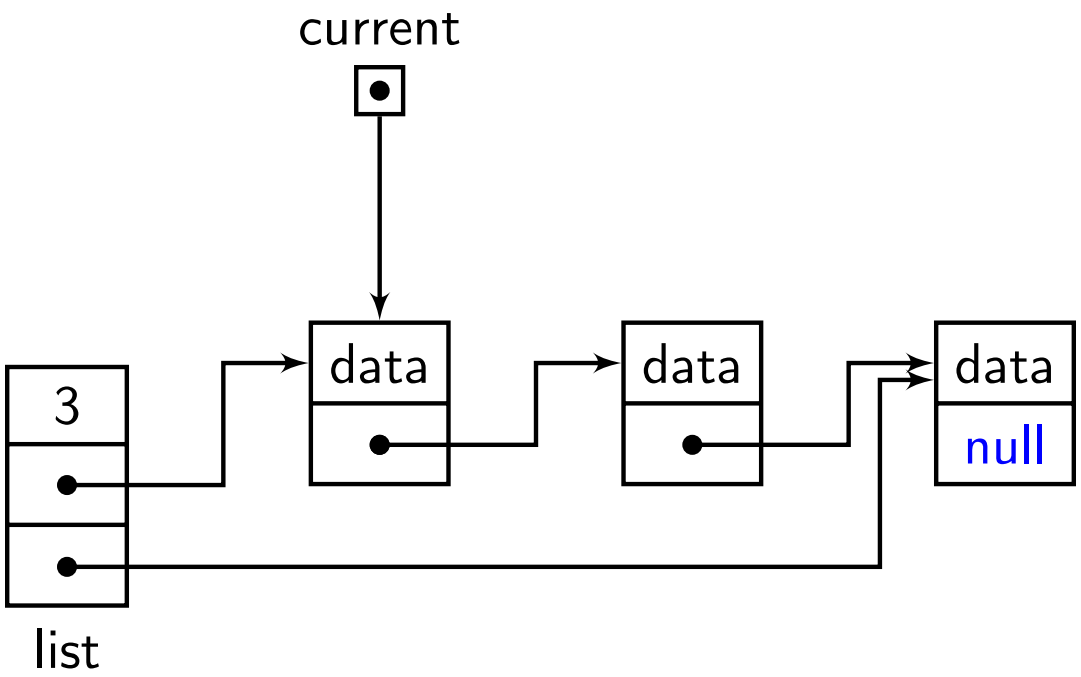
Suivant l'API, la suppression dans une liste est réalisée à partir d'un élément.



Étapes de suppression

Suppression dans une liste chaînée

Suivant l'API, la suppression dans une liste est réalisée à partir d'un élément.

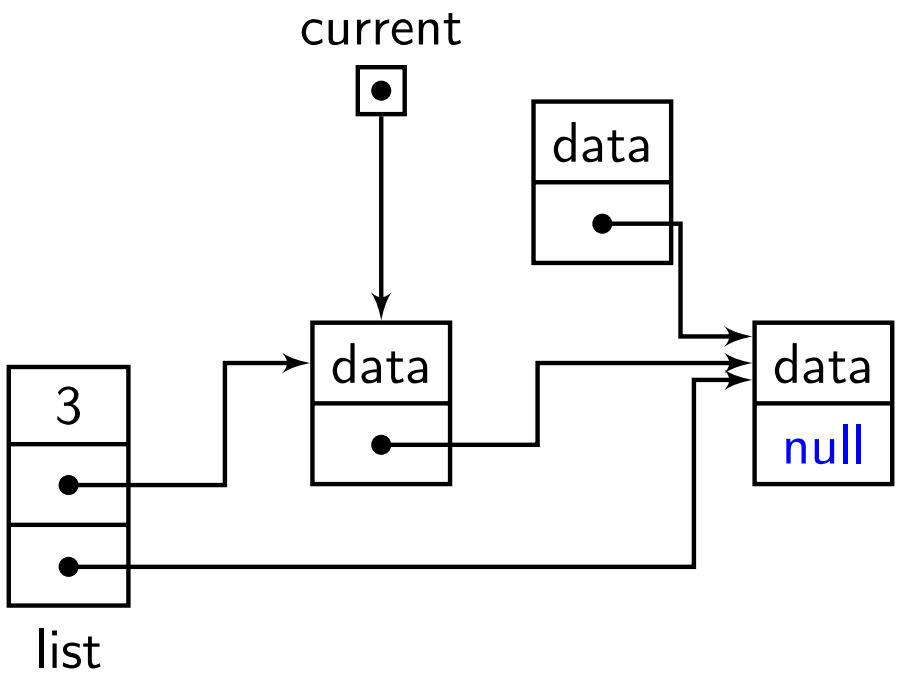


Étapes de suppression

1. On donne l'élément après lequel supprimer un nouvel élément

Suppression dans une liste chaînée

Suivant l'API, la suppression dans une liste est réalisée à partir d'un élément.

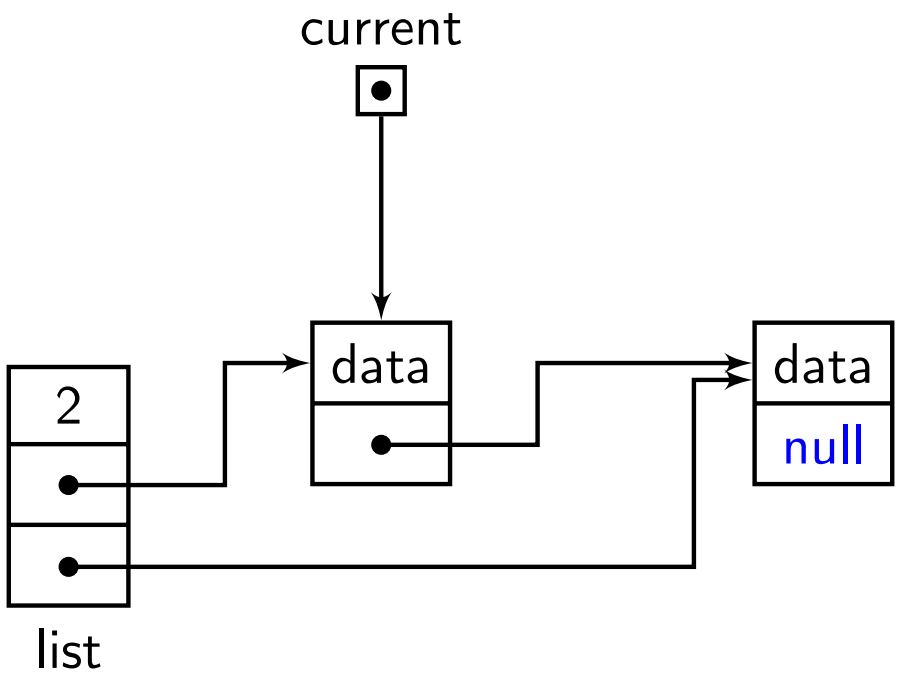


Étapes de suppression

1. On donne l'élément après lequel supprimer un nouvel élément
2. On fait le nouveau chaînage (une seule opération)

Suppression dans une liste chaînée

Suivant l'API, la suppression dans une liste est réalisée à partir d'un élément.



Étapes de suppression

1. On donne l'élément après lequel supprimer un nouvel élément
2. On fait le nouveau chaînage (une seule opération)
3. On désalloue la cellule
4. On met à jour de la taille de la liste chaînée

Suppression dans une liste chaînée

Quelques comportements particuliers

- Que se passe-t-il si l'élément après lequel supprimer est le pointeur `NULL` ?

  Suppression en tête

- Que se passe-t-il si on essaie de faire une suppression du dernier élément ?

  Pas possible, simplement code de retour égale à -1

  Note : pour supprimer le dernier élément il faut garder un pointeur sur le pénultième élément.

- Que se passe-t-il si on essaie de faire une suppression dans une liste vide ?

  Pas possible, simplement code de retour égale à -1

Complexité des opérations sur les listes

Fonction	Complexité	Commentaire
Initialisation	$\mathcal{O}(1)$	Mise de la mémoire à zéro
Destruction	$\mathcal{O}(n)$	Parcourir la liste pour supprimer les éléments
Accesseurs	$\mathcal{O}(1)$	
Insertion	$\mathcal{O}(1)$	Hyp : on sait où faire l'insertion
Suppression	$\mathcal{O}(1)$	Hyp : on sait où faire l'insertion
Recherche	$\mathcal{O}(n)$	Pas d'accès direct, il faut parcourir la liste

Remarque

La structure de liste a été pensée pour avoir des opérations avec des complexités basses.
Essentiellement car une liste est formée d'un pointeur de tête, d'un pointeur de queue, d'un champs taille.

Retour sur les types génériques

Liste chaînée générique

```
typedef struct generic_list_elmt_ {
  void *data;
  struct generic_list_elmt_  *next;
} generic_list_elmt_t;

typedef struct generic_list_ {
  int size;
  int (*compare)(const void *key1, const void *key2);
  void* (*build)(const void *data);
  void (*destroy)(void *data);
  generic_list_elmt_t *head;
  generic_list_elmt_t *tail;
} generic_list_t;
```

Conséquence :

- Comme `void*` est un pointeur vers une zone mémoire sans autre information, il faut indiquer comment créer des éléments, comment comparer des éléments et comment détruire des éléments.

Retour sur les types génériques

La principale modification concerne la fonction d'initialisation,

Fonction d'initialisation [a]

a. Le mot clef `const` indique au compilateur que la zone mémoire pointée ne doit pas changer (pas d'affectation possible sur les arguments).

```
void generic_list_init(generic_list_t *list,
            int (*compare) (const void *key1, const void *key2),
            void* (*build)(const void *data),
            void (*destroy)(void *data));
```

Dans ce contexte

- on utilise des pointeurs sur des fonctions.
- notez le prototype de ces pointeurs de fonction

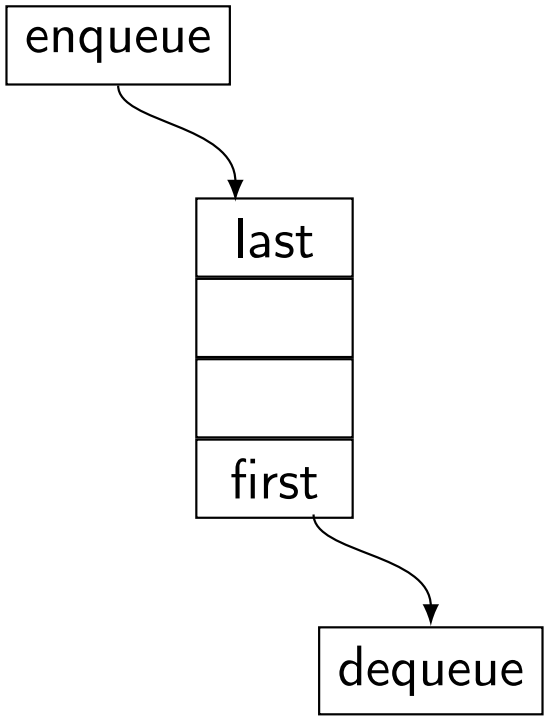
```
<type de retour> (*nom) (<types arguments>);
```

- Toutes autres fonctions suivent le même format excepté que vous devez mémoriser quel type de données est stocké ! !

Files

Structure de données : File

- Stocker des choses à traiter dans l'ordre d'arrivée ($\approx$ file d'attente).
- Comportement (*FIFO* : F*irst* I*n* F*irst* O*ut*) : premier inséré, premier retiré
- On peut consulter les éléments dans la file mais pas les retirer tant que ceux de devant sont dans la file.
- On traite les éléments dans l'ordre d'arrivée (ou d'insertion dans la file)



Applications des files

- Stocker des «transactions» à effectuer : mémoire tampon («buffer»), accès à des zones mémoires protégées.
- Ordonnancer les processus dans un OS multitâche.
- Parcours «en largeur » d'arbres ou de graphes (voir plus tard les cours sur les arbres et les graphes).
- Services client/serveur (Web).

Implémentations des files

Plusieurs façon de faire :

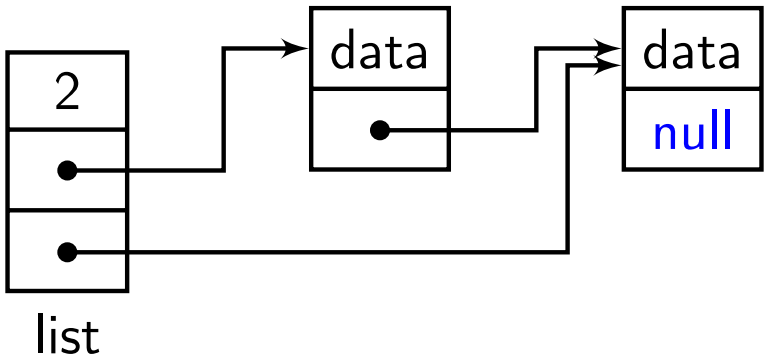
- A l'aide de tableaux de manière circulaire

```
struct queue {
  unsigned int max_nb;     /* Nombre max d'elements. */
  unsigned int cur_nb;     /* Nombre actuel d'elements. */
  unsigned int first;      /* Indice du premier element. */
  int *data;
};
```

  Remarque : On doit mettre en œuvre des gestions des dépassements de capacité (exception/realloc)

- A l'aide d'une liste chaînée comme dans la bibliothèque `libin103`

Implémentation des files dans `libin103`



Une file est une liste

```
typedef integer_list_t integer_queue_t;
```

MAIS avec une API différente

API des files – 1

API des files – Création/Destruction

```
void integer_queue_init (integer_queue_t* queue);

void integer_queue_destroy (integer_queue_t* queue);
```

API des files – Accesseurs

```
int integer_queue_size (integer_queue_t* queue);

int integer_queue_peek(integer_queue_t* queue);
```

- peek, fonction pour récupérer (sans supprimer) le premier élément (le plus vieux) de la file

API des files – 2

API des files – Insertion/Suppression

```
int integer_queue_enqueue(integer_queue_t* queue, int data);

int integer_queue_dequeue(integer_queue_t* queue, int* data);
```

- enqueue, fonction pour insérer un nouvel élément dans la file
- dequeue, fonction pour supprimer le premier élément (le plus vieux) dans la file

Complexité des opérations sur les files

Fonction	Complexité	Commentaire
Initialisation	$\mathcal{O}(1)$	Mise de la mémoire à zéro
Destruction	$\mathcal{O}(n)$	Pareil que pour les listes
Accesseurs	$\mathcal{O}(1)$	
`Enqueue`	$\mathcal{O}(1)$	Équivalent à insertion en fin de la liste
`Dequeue`	$\mathcal{O}(1)$	Équivalent à suppression en début de la liste

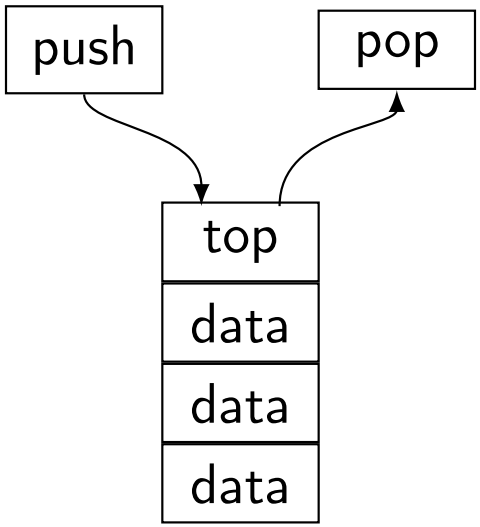
Remarque

La structure de liste a été pensée pour mettre en œuvre facilement des files d'où les classes de complexité basses

Piles

Structure de données : Pile

- Comment dans la vie courante : stocker des choses comme une pile d'assiettes.
- Comportement (*LIFO* : L*ast* I*n* F*irst* O*ut*) : dernier déposé, premier retiré
- On ne peut pas consulter les éléments sous le sommet de la pile.



Applications des piles

- Garder trace d'un historique de traitements (voir plus tard les cours sur les arbres et les graphes)
- Garder trace de traitements non complétés («bracktracking») : classique en programmation par contraintes

- Base de nombreuses machines virtuelles (Java par exemple).

Exemple, notation polonaise inversée (RPN)

- Notation postfixe "1 2 + 5 ×"
- Opérateurs après les opérandes (comme les anciennes calculatrices HP)
- Opérandes sur la pile, opérateur opérant sur les éléments de la pile.

- . . . Même structure que la pile d'exécution d'un programme C. . .

Implémentations des piles

Plusieurs façon de faire :

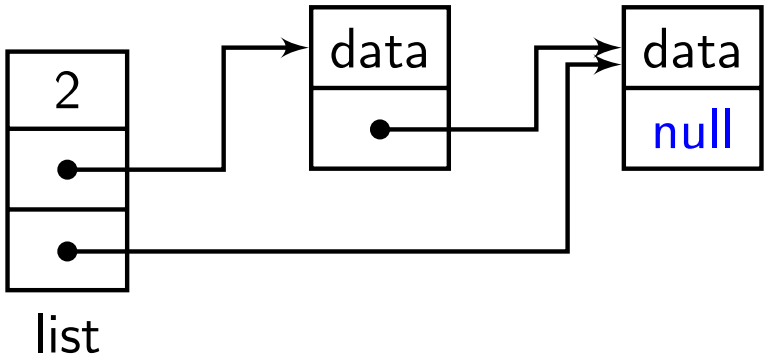
- A l'aide de tableaux et un pointeur de pile indiquant la prochaine case libre

```
#define MAX_SIZE 64
struct stack {
    unsigned int sp;      /* Pointeur de pile. */
    int data[MAX_SIZE];    /* Memoire de pile. */
};
```

  Remarque : On doit mettre en œuvre des gestions des dépassements de capacité (exception/realloc)

- A l'aide d'une liste chaînée comme dans la bibliothèque `libin103`

Implémentation des piles



Une pile est une liste

```
typedef integer_list_t integer_stack_t;
```

MAIS avec une API différente

API des piles – 1

API des piles – Création/Destruction

```
void integer_stack_init (integer_stack_t* stack);

void integer_stack_destroy (integer_stack_t *stack);
```

API des piles – Accesseurs

```
int integer_stack_size (integer_stack_t * stack);

int integer_stack_peek(integer_stack_t * stack);
```

- peek, fonction pour récupérer (sans supprimer) le premier élément de la pile (le sommet, la valeur la plus récente)

API des piles – 2

API des piles – Insertion/Suppression

```
int integer_stack_push(integer_stack_t *stack, int data);

int integer_stack_pop(integer_stack_t *stack, int* data);
```

- push, fonction pour insérer un nouvel élément au sommet de la file
- pop, fonction pour supprimer un le sommet de la file

Complexité des opérations sur les piles

Fonction	Complexité	Commentaire
Initialisation	$\mathcal{O}(1)$	Mise de la mémoire à zéro
Destruction	$\mathcal{O}(n)$	Même cas que les listes
Accesseurs	$\mathcal{O}(1)$	
`Push`	$\mathcal{O}(1)$	Équivalent à insertion en début de la liste
`Pop`	$\mathcal{O}(1)$	Équivalent à suppression en début de la liste

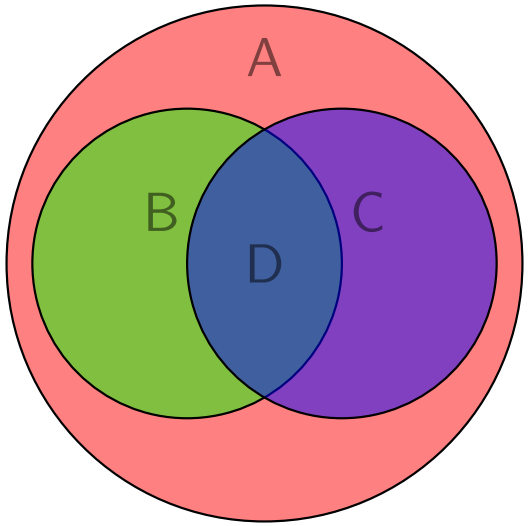
Remarque

La structure de liste a été pensée pour mettre en œuvre facilement ces piles d'où les classes de complexité basse.

Ensembles

Structure de données : ensembles

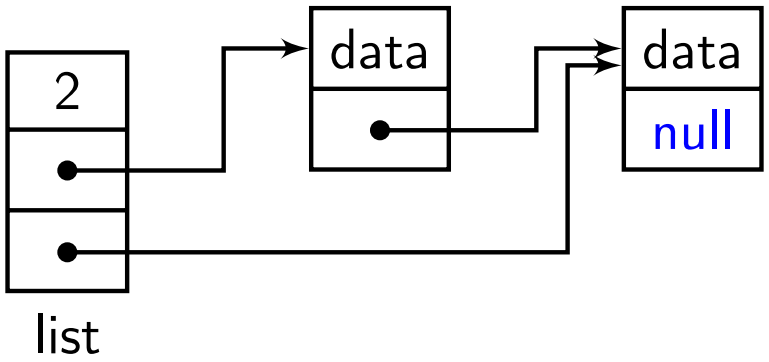
- Collection d'éléments sans multi occurrence
- Appliquer des opérations ensemblistes : union, intersection, différence



Quelques applications des ensembles

- Corrélation de données
- Couverture d'ensembles (cf TP)
- Opérations mathématiques sur les ensembles
- Les algèbres relationnelles comme pour le langage SQL

Implémentation des ensembles



Un ensemble est une liste

```
typedef integer_list_t integer_set_t;
```

mais avec une API différente

API des ensembles – 1

API des ensembles – Création/Destruction

```
void integer_set_init(integer_set_t *set);

void integer_set_destroy (integer_set_t *set);
```

API des ensembles – Accesseurs

```
int integer_set_size(integer_set_t*);
```

API des ensembles – Prédicats

```
bool integer_set_is_member(integer_set_t*, int);

bool integer_set_is_subset(integer_set_t*, integer_set_t*);

bool integer_set_is_equal(integer_set_t*, integer_set_t*);
```

API des ensembles – 2

API des ensembles – Insertion/Suppression

```
int integer_set_insert(integer_set_t *set, int data);

int integer_set_remove(integer_set_t *set, int *data);
```

- `insert`, fonction pour insérer un nouvel élément au dans l'ensemble en garantissant l'unicité

API des ensembles – Opérations ensemblistes

```
int integer_set_union(integer_set_t *setu,
    integer_set_t*, integer_set_t*);

int integer_set_intersection(integer_set_t *seti,
    integer_set_t*, integer_set_t*);

int integer_set_difference(integer_set_t *setd,
    integer_set_t*, integer_set_t*);
```

Intuition des algorithmes

On considère deux ensembles s_1 et s_2 de cardinalité respectives n_1 et n_2.

- Insertion : ajout d'un nouvel élément si pas déjà présent (test d'appartenance)
- Union : insérer les éléments de s_1 dans s_2
- Intersection : pour chaque élément de s_1, on vérifie sa présence dans s_2, si oui insertion
- Différence : pour chaque élément de s_1, on vérifie sa présence dans s_2, si non insertion
- Test sous ensemble : vérifie que tous les éléments de s_1 sont dans s_2
- Test égalité : tous les éléments de s_1 sont dans s_2 et tous les éléments de s_2 sont dans s_1

Complexité des opérations sur les ensembles

Fonction	Complexité	Commentaire
Initialisation	$\mathcal{O}(1)$	Mise de la mémoire à zéro
Destruction	$\mathcal{O}(n)$	Pareil que pour les listes
Accesseurs	$\mathcal{O}(1)$	
Test appartenance	$\mathcal{O}(n)$	Parcours de liste
Test sous ensemble	$\mathcal{O}(n_1 n_2)$	Parcours des deux listes
Test égalité	$\mathcal{O}(n_1 n_2)$	Parcours des deux listes (2 fois)
`Insert`	$\mathcal{O}(n)$	Vérification de l'appartenance
`Remove`	$\mathcal{O}(n)$	Recherche de l'élément à supprimer
Union	$\mathcal{O}(n_1 n_2)$	Parcours des deux listes
Intersection	$\mathcal{O}(n_1 n_2)$	Parcours des deux listes
Différence	$\mathcal{O}(n_1 n_2)$	Parcours des deux listes

Remarque

La structure de liste permet une mise en œuvre facile des ensembles mais avec une complexité importante. On peut faire mieux si on considère une structure sous jacente arborescente, cf plus tard dans le cours.

Une nouvelle organisation des données

Jusqu'à présent, des structures de données linéaires ont été étudiées :

- tableau : taille fixe, accès directe
- liste chaînée : taille variable, accès indirect
- pile : comportement LIFO
- file : comportement FIFO
- ensemble : unicité des éléments

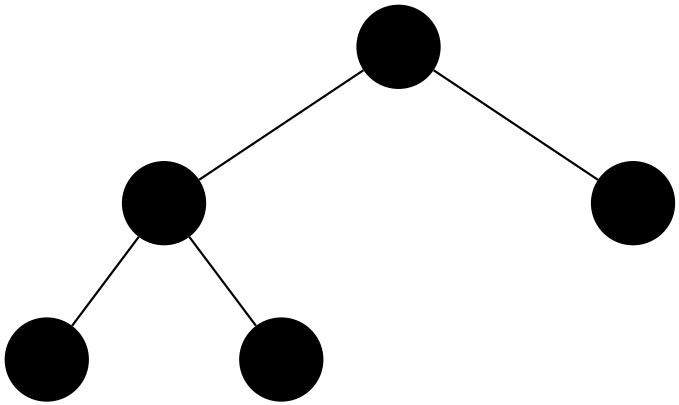
L'organisation des données dans ces structures est sans relation mathématique particulière.

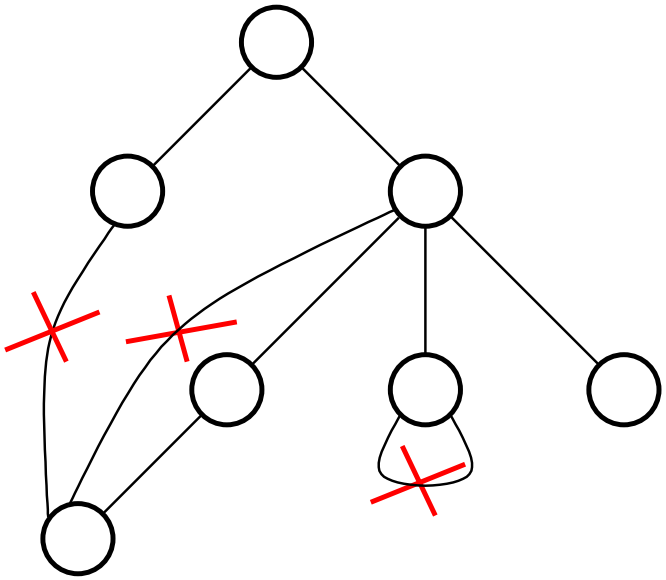
Conséquence, la recherche dans une liste chaînée a une complexité $\mathcal{O}(n)$.

Les arbres sont une structure de données associées à une relation d'ordre.

Arbres

Structure arborescente





- Structure arborescente avec une relation parent/enfant
- Des nœuds, des arcs
- mais pas de cycle, pas de raccourci (saut de générations)

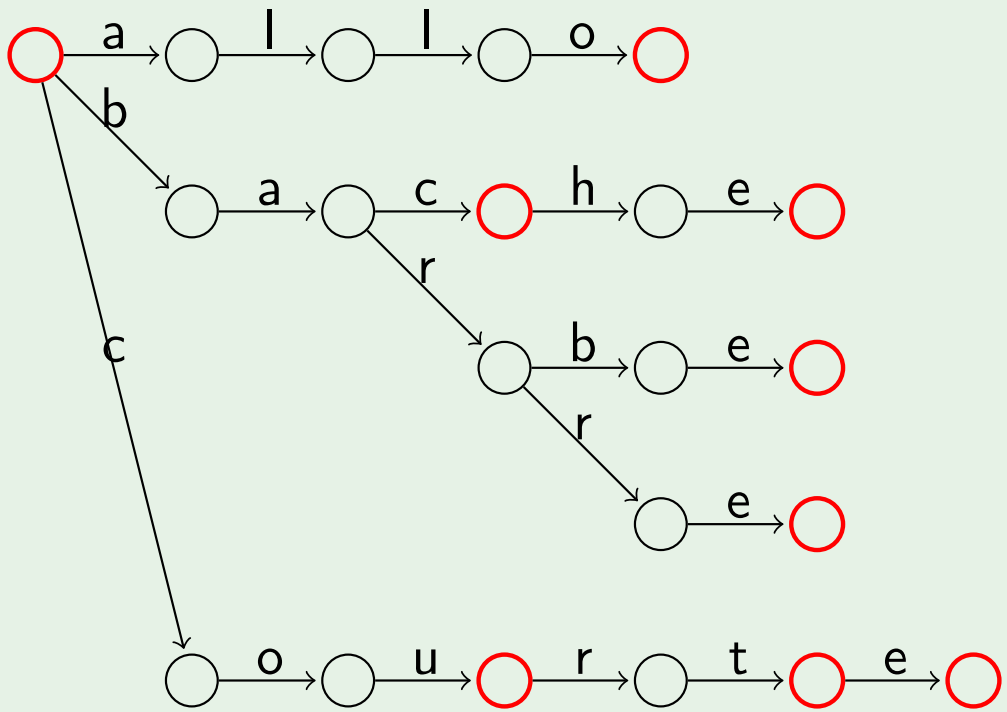
Exemple : dictionnaire

Dictionnaire représenté par un arbre préfixe (en anglais *trie*)

- Permet le partage de préfixes communs,
- Préfixes partagés d'où nécessité de marquer les fins de mots

Exemple

Les mots { allo, bac, bache, barbe, barre, cou, court, courte } sont représentés par



Exemple : système de fichiers (simple)

```
/
|-- bin
|   |-- alsaunmute
|   |-- arch
|   |-- awk
|   `-- zcat
|-- boot
|   |-- config-2.6.40.6-0.fc15.i686
|   |-- grub
|   |   |-- device.map
|   |   |-- e2fs_stage1_5
|   |   |-- ufs2_stage1_5
|   |   |-- vstafs_stage1_5
|   |   `-- xfs_stage1_5
|   |-- initramfs-2.6.40.6-0.fc15.i686.img
|   `-- vmlinuz-2.6.43.2-6.fc15.i686
|-- cgroup
|-- dev
|   |-- autofs
```

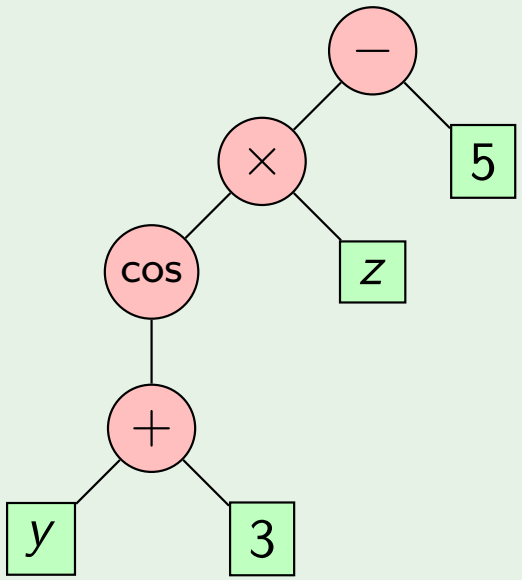
- Racine : /
- Nœuds internes : répertoires.
- Feuilles : fichiers.
- (On ne considère pas les « raccourcis » – ou « liens »).

Exemple : arbre de syntaxe

Expressions arithmétiques :

- Constantes.
- Opérateurs binaires entre expressions arithmétiques.
- Appel de fonction avec des expressions arithmétiques en arguments.
- Variables.

Exemple : $\cos(y + 3) \times z - 5$

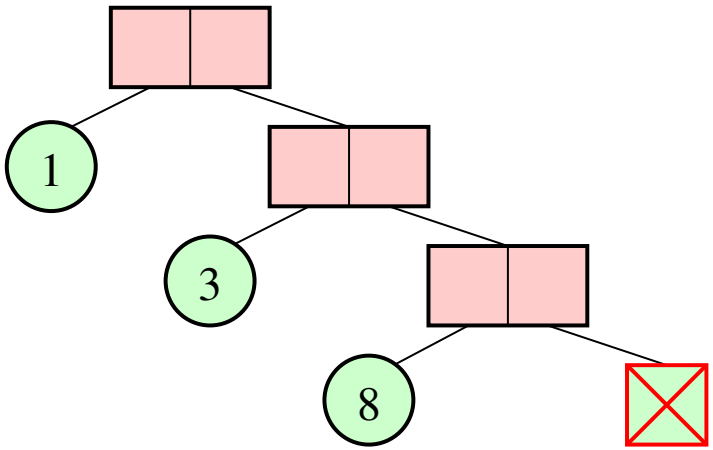


Permet de calculer

- évaluation
- dérivation
- simplification
- compilation, ...

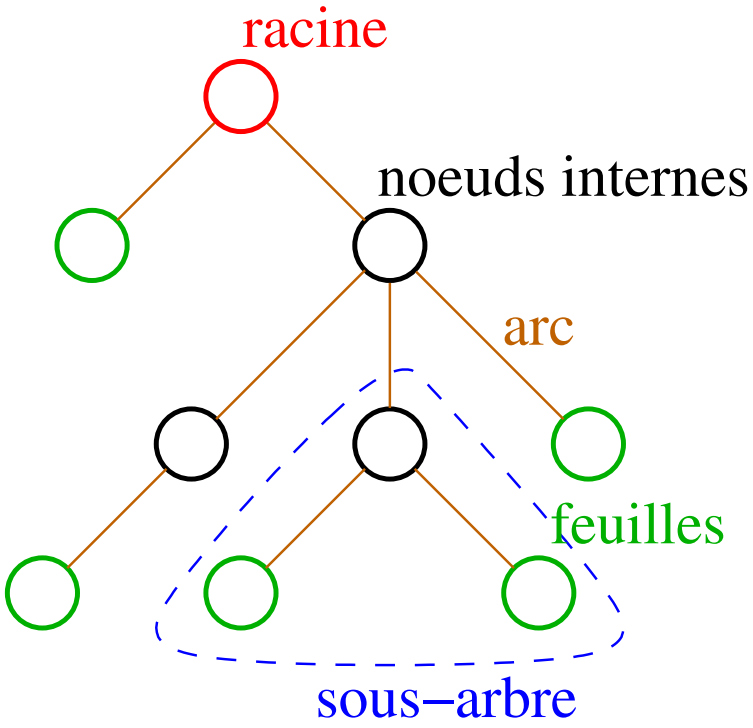
Exemple : listes simplement chaînées

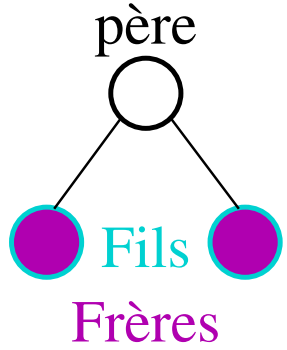
- Une liste : [1, 3, 8]
- Rappel : type inductif
  `list` ::=
  `Nil` | `List of int` $*$ `list`
- Longueur de la liste = hauteur de l'arbre



Arbres : vocabulaire - 1

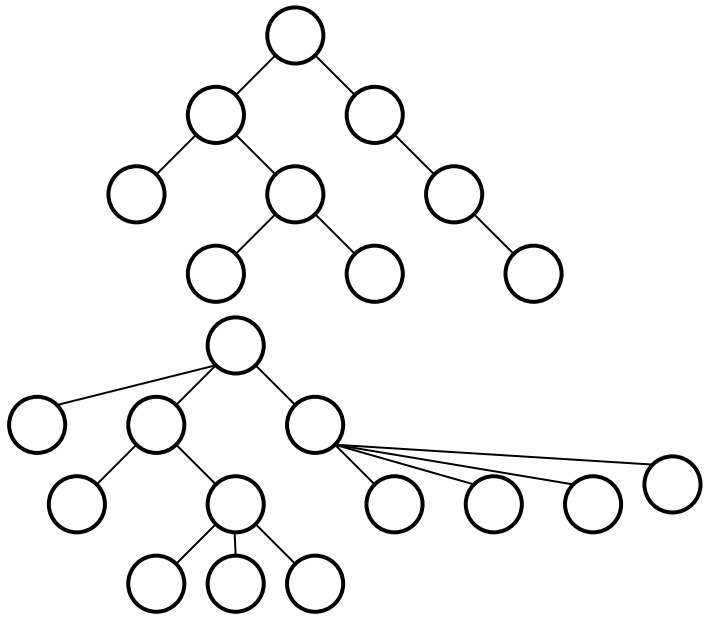
- Nœud (*node*) :
  - Racine (*root*)
  - Nœud interne
  - Feuille (*leaf*)
- Arc (*edge*)
- Sous-arbre (*subtree*)
- Parenté :
  - Père (*parent*)
  - Fils (*child*)
  - Frère (*sibling*)





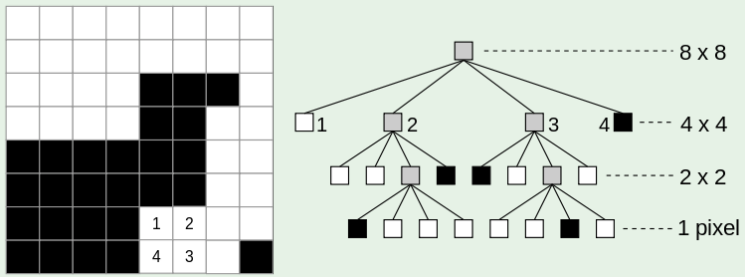
Arbres : vocabulaire - 2

- Degré d'un nœud : nombre de fils
- Arité d'un arbre : nombre max de fils
  - Pas forcément homogène sur tous les nœuds de l'arbre
- Arbre binaire : arité 2
- Quadtree : arité 4
- Octree : arité 8
- Autrement arbre n-aire



Compression d'images avec quadtree [a]

a. Crédit `https://fr.wikipedia.org/wiki/Quadtree`



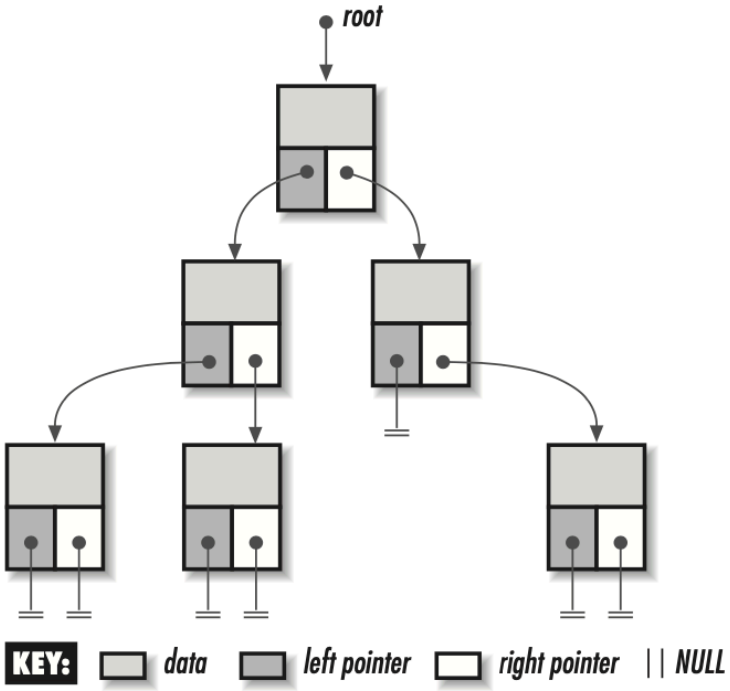
Arbres binaires

Arbres binaires : implémentation

Type inductif utilisé dans les langages fonctionnels (Ocaml, Haskel, ...)

- $\texttt{tree} ::= \texttt{Empty} \mid \texttt{Node of tree} * \texttt{elem} * \texttt{tree}$

Comme pour les listes, les pointeurs sont utilisés pour faire un chaînage [2]



Définition d'un nœud

```
typedef struct integer_bitreenode_ {
  int data;
  struct integer_bitreenode_ *left;
  struct integer_bitreenode_ *right;
} integer_bitreenode_t;
```

Définition d'un arbre

```
typedef struct integer_bitree_ {
  int size;
  integer_bitreenode_t *root;
} integer_bitree_t;
```

2. Crédit image : O'Reilly 'Mastering Algorithms with C'

API des arbres binaires – 1

API Création/Destruction

```
void integer_bitree_init(integer_bitree_t*);

void integer_bitree_destroy(integer_bitree_t*);
```

API Accesseurs

```
int integer_bitree_size(integer_bitree_t*);

integer_bitreenode_t* integer_bitree_root(integer_bitree_t*);

int integer_bitree_data(integer_bitreenode_t*);

integer_bitreenode_t* integer_bitree_left(integer_bitreenode_t*);

integer_bitreenode_t* integer_bitree_right(integer_bitreenode_t*);
```

API des arbres binaires – 2

API Prédicats

```
bool integer_bitree_is_eob(integer_bitreenode_t*);

bool integer_bitree_is_leaf(integer_bitreenode_t*);
```

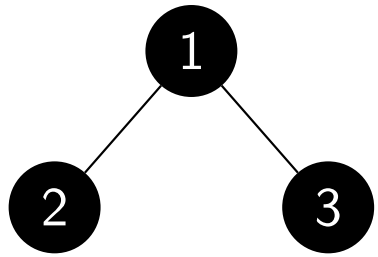
API Insertion/Suppression

```
int integer_bitree_ins_left(integer_bitree_t*,
                            integer_bitreenode_t*, int);

int integer_bitree_ins_right(integer_bitree_t*,
                             integer_bitreenode_t *, int);

void integer_bitree_rem_left(integer_bitree_t*,
                             integer_bitreenode_t*);

void integer_bitree_rem_right(integer_bitree_t*,
                              integer_bitreenode_t*);
```

Exemple : un petit arbre binaire

Code source

```
int main () {
  int code;
  integer_bitreenode_t* root;
  integer_bitree_t tree;
  integer_bitree_init (&tree);

  code = integer_bitree_ins_left (&tree, NULL, 1);
  printf ("Root takes value 1\n");

  root = integer_bitree_root (&tree);
  code = integer_bitree_ins_left (&tree, root, 2);
  printf ("Root->left takes value 2\n");
  code = integer_bitree_ins_right (&tree, root, 3);
  printf ("Root->right takes value 3\n");

  integer_bitree_destroy (&tree);

  return EXIT_SUCCESS;
}
```



Complexité des opérations sur les arbres binaires

Fonction	Complexité	Commentaire
Initialisation	$\mathcal{O}(1)$	Mise de la mémoire à zéro
Destruction	$\mathcal{O}(n)$	Parcourir tous les nœuds pour supprimer les éléments
Accesseurs	$\mathcal{O}(1)$	
Insertion	$\mathcal{O}(1)$	Hyp : on sait où faire l'insertion
Suppression	$\mathcal{O}(n)$	Hyp : on sait où faire la suppression mais il faut éliminer tous les nœuds du sous-arbre.
Recherche	$\mathcal{O}(n)$	les valeurs sont stockées sans ordre particulier, il faut donc tout parcourir.

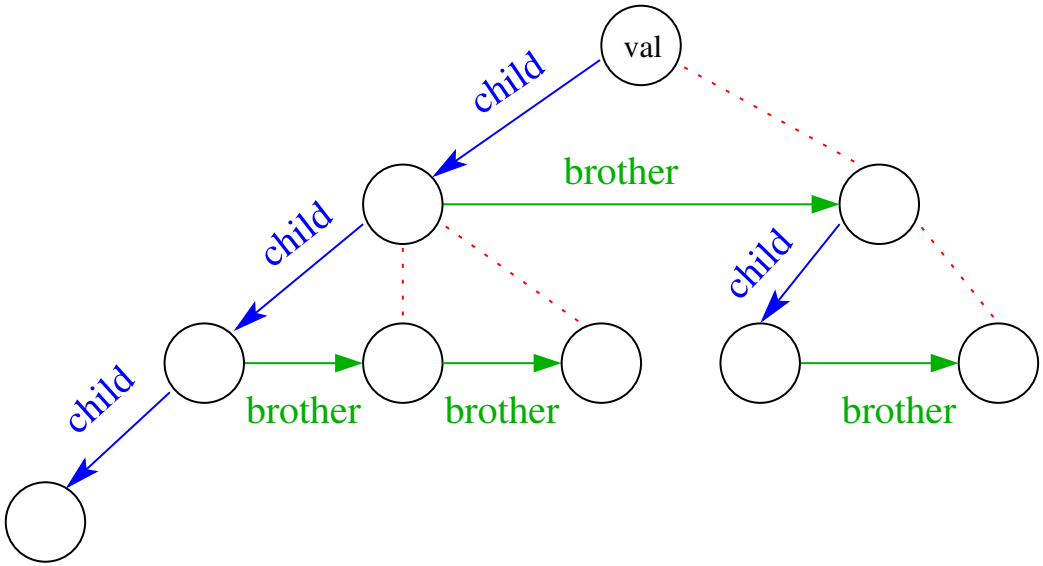
Remarque

La structure d'arbre binaire permet de construire un arbre facilement mais n'est pas pensée pour chercher des éléments, cf plus loin.

Arbre n-aire

Cas général, un nombre quelconque de fils, un nœud est

- une donnée.
- une liste chaînée de tous les fils (frères).
- le parent a un pointeur sur son premier fils.



Code source

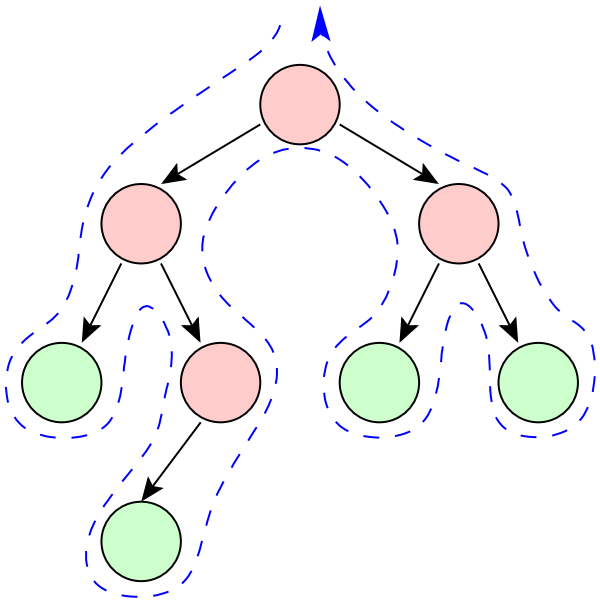
```
struct node {
  int data;
  struct node* child;
  struct node* brothers;
};
```

Remarque : on peut aussi utiliser un tableau pour stocker les pointeurs des enfants mais cela nécessite de fixer l'arité de l'arbre au départ.

Parcours d'arbres

Arbres binaires : parcours en profondeur

- En Anglais : *Depth First Search* (DFS).
- Descente au plus profond en commençant par le fils gauche (ou droit).
- Parcours récursif :
  - À chaque nœud, on applique la descente sur le fils gauche.
  - Une fois le sous-arbre gauche exploré, on explore le fils droit.
  - Une fois le sous-arbre droit exploré, on remonte et applique le parcours.
  - Fin : fin de l'exploration du sous-arbre droit.



Function
```
dfs(integer_bitreenode_t* n) :
    if n != NULL then
        dfs(n->left);
        dfs(n->right);
    end
```

Remarque : pseudo-code du parcours sans traitement !

Parcours en profondeur

Le traitement de la valeur du nœud courant peut se faire à des moments différents durant le parcours

- Ordre préfixe : traitement, puis visite gauche, visite droit
- Ordre infixe : visite gauche, traitement, visite, droit
- Ordre postfixe : visite gauche, visite droit, traitement

```
Function dfs(integer_bitreenode_t* n) :
    if n != NULL then
        do_something(n->data)
        dfs(n->left)

        dfs(n->right)

    end
```

Parcours en profondeur

Le traitement de la valeur du nœud courant peut se faire à des moments différents durant le parcours

- Ordre préfixe : traitement, puis visite gauche, visite droit
- Ordre infixe : visite gauche, traitement, visite, droit
- Ordre postfixe : visite gauche, visite droit, traitement

```
Function dfs(integer_bitreenode_t* n) :
    if n != NULL then
        dfs(n->left)
        do_something(n->data)
        dfs(n->right)
    end
```

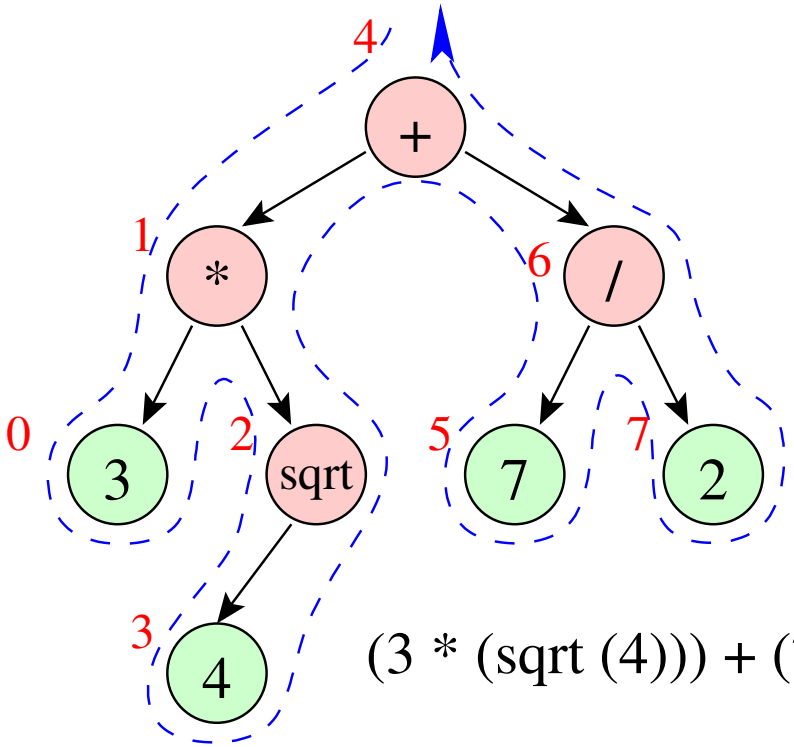
Parcours en profondeur

Le traitement de la valeur du nœud courant peut se faire à des moments différents durant le parcours

- Ordre préfixe : traitement, puis visite gauche, visite droit
- Ordre infixe : visite gauche, traitement, visite, droit
- Ordre postfixe : visite gauche, visite droit, traitement

```
Function dfs(integer_bitreenode_t* n) :
    if n != NULL then
        dfs(n->left)
        dfs(n->right)
        do_something(n->data)
    end
```

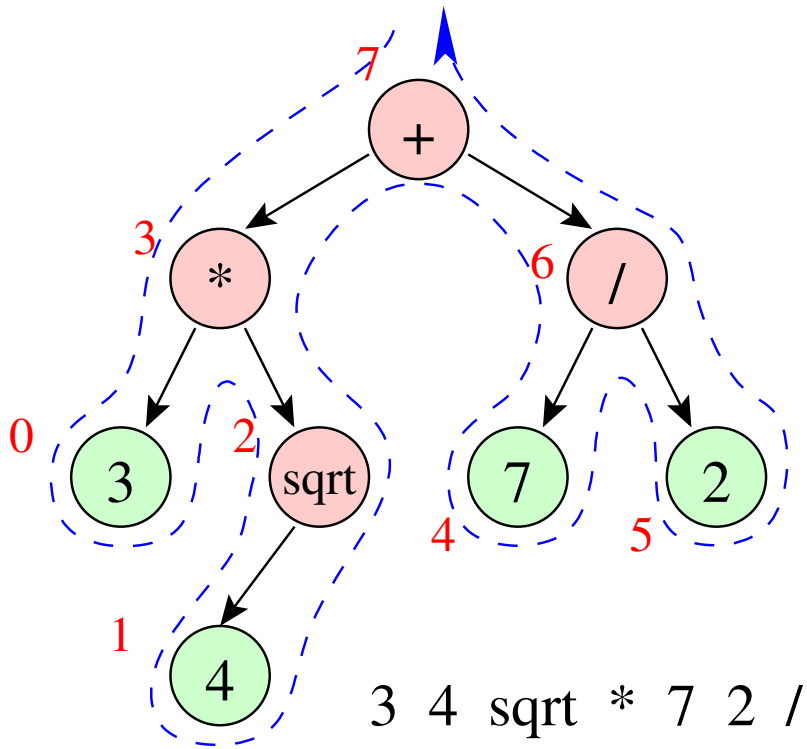
Exemple : parcours infixe



(3 * (sqrt (4))) + (7 / 2) (= 9.5)

Remarque en rouge l'ordre de traitement des nœuds

Exemple : parcours postfixe



3 4 sqrt * 7 2 / + (= 9.5)

Remarque en rouge l'ordre de traitement des nœuds

Implémentation des parcours d'arbres dans `libin103`

Trois fonctions sont accessibles dans le fichier `bitreealg.h`

Prototypes

```
int integer_preorder(integer_bitreenode_t*, integer_list_t*);

int integer_inorder(integer_bitreenode_t*, integer_list_t*);

int integer_postorder(integer_bitreenode_t*, integer_list_t*);
```

- Le premier argument est le nœud de l'arbre à partir duquel il faut effectuer le parcours.
- Le second argument est une liste des valeurs des nœuds visités dans l'ordre choisi.

Remarque

Pour effectuer un traitement sur les nœuds, il faut coder une version spécialisée du parcours voulu.

Complexité des fonctions de parcours

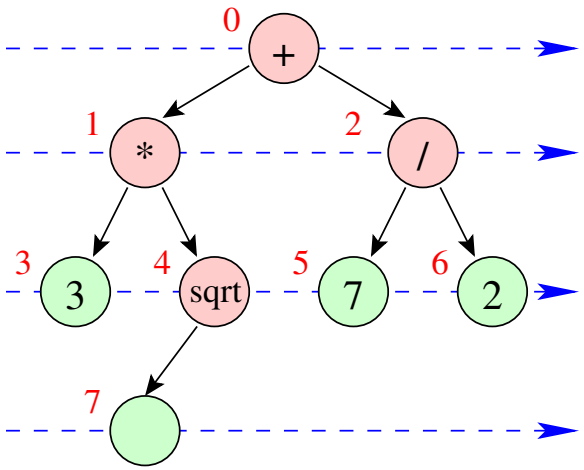
Fonction	Complexité	Commentaire
Préfixe	$\mathcal{O}(n)$	Il faut parcourir tous les nœuds les éléments
Infixe	$\mathcal{O}(n)$	Il faut parcourir tous les nœuds les éléments
Postfixe	$\mathcal{O}(n)$	Il faut parcourir tous les nœuds les éléments

Remarque

Les parcours sont exhaustifs d'où le coût linéaire.

Arbres binaires : parcours en largeur

- En Anglais : *Breadth First Search* (BFS).
- Parcours « par niveaux » (croissants), visite tous les nœuds de « même profondeur ».
- Utilisation d'une structure de données file :
  On extrait un nœud, on le traite, on insère tous ses fils.



Remarque en rouge l'ordre de traitement des nœuds

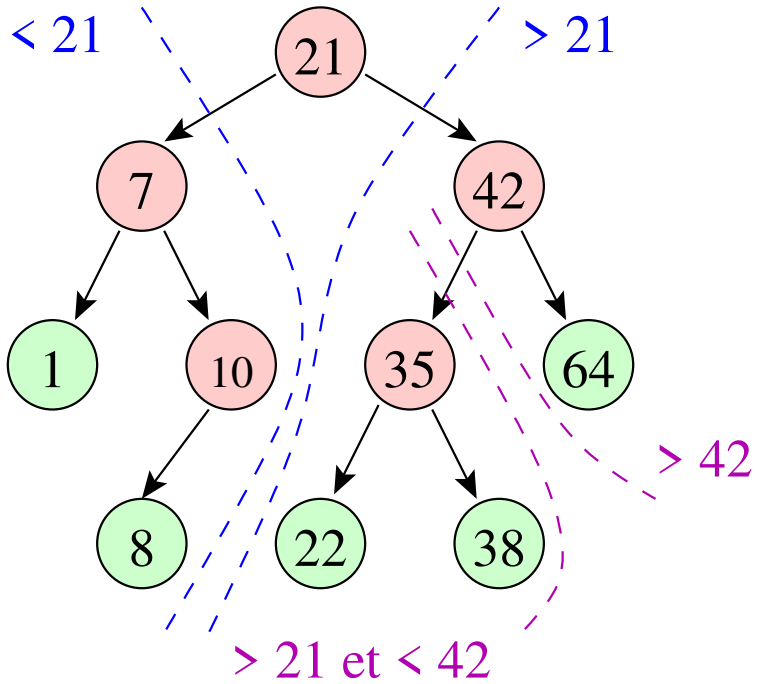
```
Function bfs(integer_bitreenode_t* n) :
    generic_queue_enqueue(q, n);
    while generic_queue_size(q) > 0 do
        generic_queue_dequeue (q, tmp);
        do_something(tmp->data);
        if tmp->left != NULL then
            generic_queue_enqueue(q, n->left);
        end
        if tmp->right != NULL then
            generic_queue_enqueue(q, n->right);
        end
    end
```

Arbres binaires de recherche

Arbres Binaires de Recherche (ABR)

Structure d'arbre binaire (*Binary Search Tree* (BST)) telle que :

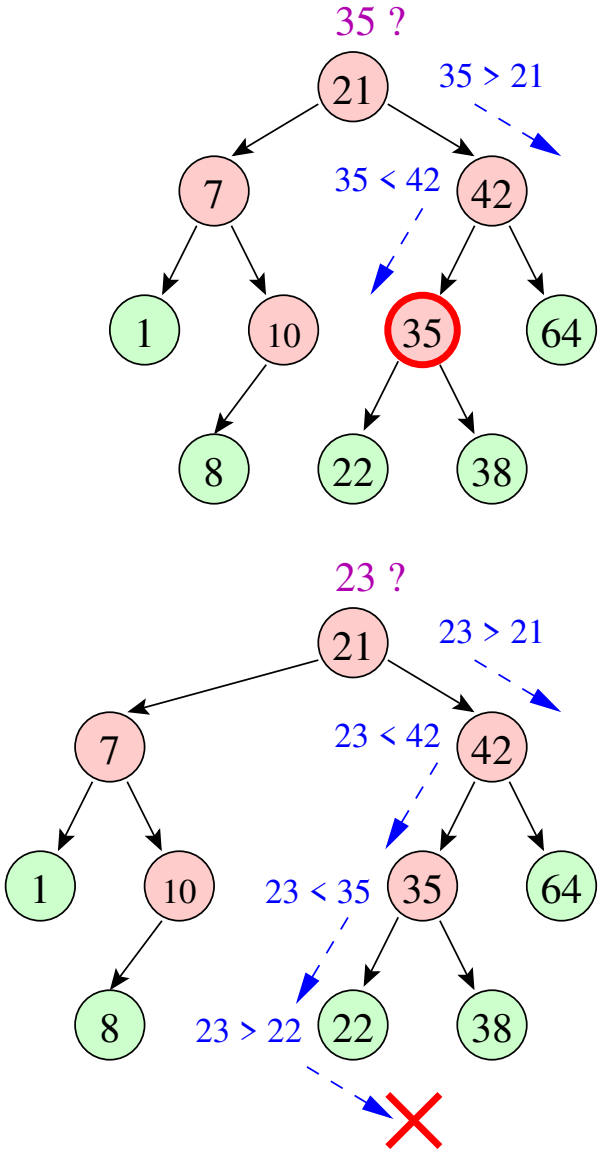
- Chaque nœud contient une clef.
- La clef d'un nœud est supérieure à celle de son fils gauche.
  Inductivement supérieur à celles du sous-arbre gauche.
- La clef d'un nœud est inférieure à celle de son fils droit.
  Inductivement inférieur à celles du sous-arbre droit.



Intérêts des ABR

- Stockage en $\mathcal{O}(n)$
- Recherche en $\mathcal{O}(h)$ (h hauteur de l'arbre).
- Insertion en $\mathcal{O}(h)$
- Suppression en $\mathcal{O}(h)$
- En moyenne $h = \mathcal{O}(\log(n))$ mais dans le pire cas $h = n$:
  $\rightarrow$ intérêt d'avoir des arbres « équilibrés » (cf plus loin)

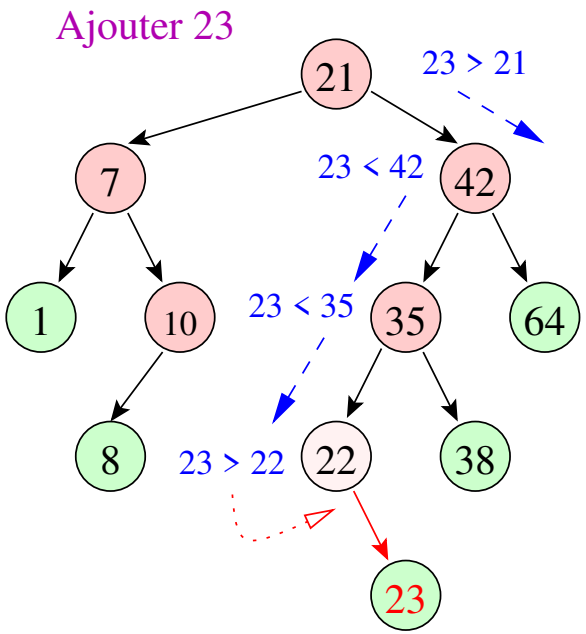
ABR : opération de recherche



Pseudo-code

```
Function lookfor(abr_t* n, int v) :
    if n == NULL then
        return false;
    end
    if v == n->data then
        return true;
    end
    if v < n->data then
        return lookfor(n->left, v);
    end
    if v > n->data then
        return lookfor(n->right, v);
    end
```

ABR : opération d'insertion



On cherche où insérer puis on créer un nouveau nœud.

Pseudo-code

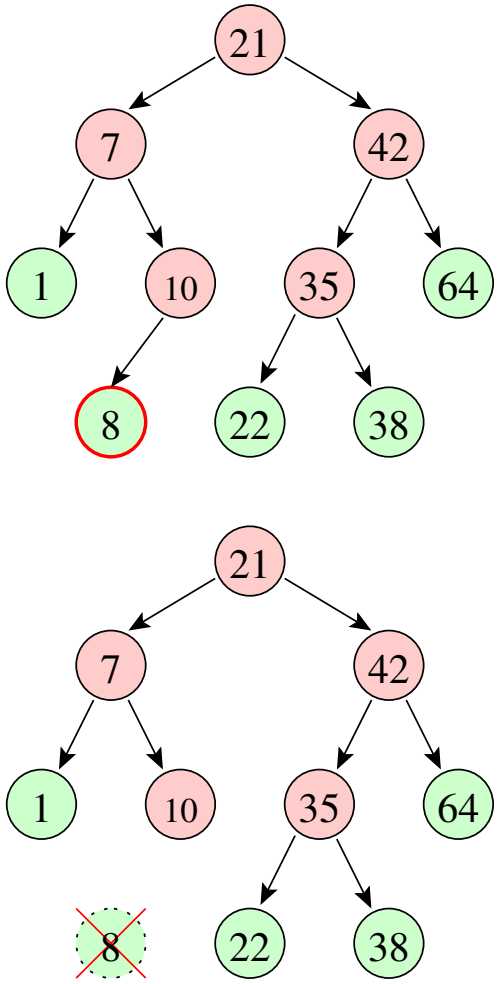
```
Function insert(abr_t* n, int v) :
    if v < n->data then
        if n->left != NULL then
            insert(n->left, v);
        else
            n->left = node(v);
        end
    else if v > n->data then
        if n->right != NULL then
            insert(n->right, v);
        else
            n->right = node(v);
        end
    else
        /* Do nothing                    */
    end
```

ABR : opération de suppression

On commence par rechercher le nœud à supprimer dans l'arbre, 3 cas dont 2 simples à gérer

- Suppression d'une feuille ($\rightarrow$ facile).
- Suppression d'un nœud avec 1 seul fils ($\rightarrow$ facile).
- Suppression d'un nœud avec 2 fils ($\rightarrow$ plus compliqué).

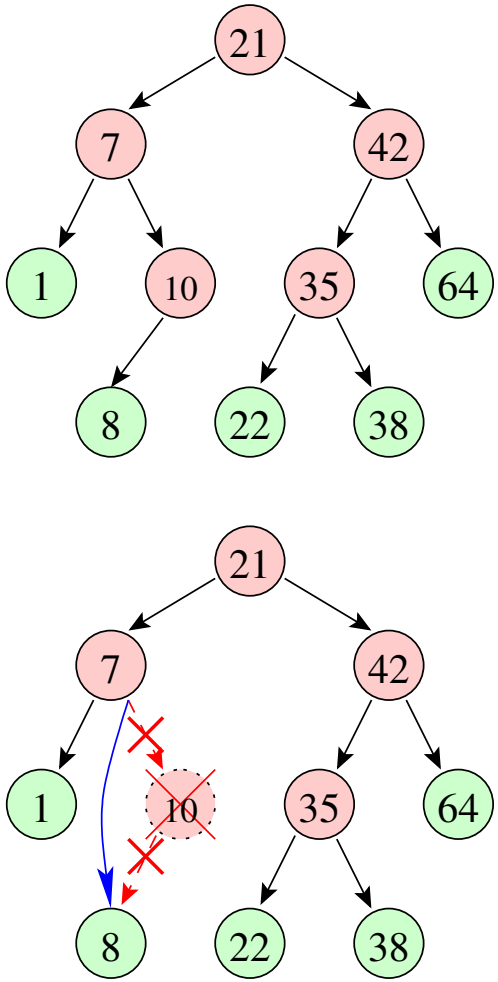
ABR : opération de suppression (feuille)



Suppression du nœud 8 . . .

- On cherche le nœud dans l'arbre
- Mettre le pointeur gauche du nœud 10 à `NULL`.
- Libérer la mémoire occupée par le nœud 8.

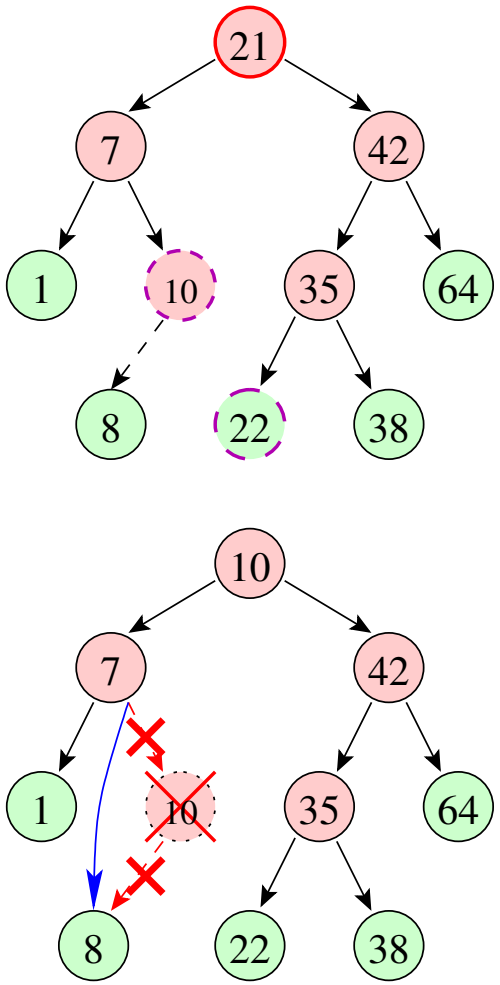
ABR : opération de suppression (1 fils)



Suppression du nœud 10 :

- On cherche le nœud dans l'arbre
- Faire pointer son père (7) vers son fils (8).
- Libérer la mémoire occupée par le nœud 10.

ABR : opération de suppression (2 fils)



Suppression du nœud 21 :

- On cherche le nœud dans l'arbre
- Remplacement par le minimum du fils droit ou le max du fils gauche.

  Successeur ou prédécesseur le plus proche.
- Choisir un candidat.
- On recopie sa clef et ses données pour remplacer le nœud 21.
- Ensuite, supprimer le nœud 10

  10 étant « le max », il n'a pas de fils droit,

  $\Rightarrow$ tombe dans un des 2 cas précédents.

Arbres binaires de recherche équilibrés

Importance des arbres équilibrés

Rappel

La complexité des opérations sur les ABR est majoritairement dans la classe $\mathcal{O}(h)$ avec h la hauteur de l'arbre.
Et dans le pire cas, $h = n$, avec n le nombre d'éléments.

Il existe plusieurs types d'ABR qui incorporent des opérations d'équilibrage

- les AVL (du nom des auteurs Adel'son, Vel'skii and Landis)
- les arbres rouges-noirs

La bibliothèque `libin103` met en œuvre les arbres AVL.

Arbres AVL

Les AVL sont des ABR dont

- la hauteur du sous-arbre gauche et la hauteur du sous-arbre droit ne diffère pas plus de 1.
- Nœuds comporte un indicateur d'équilibre (champs `factor`).

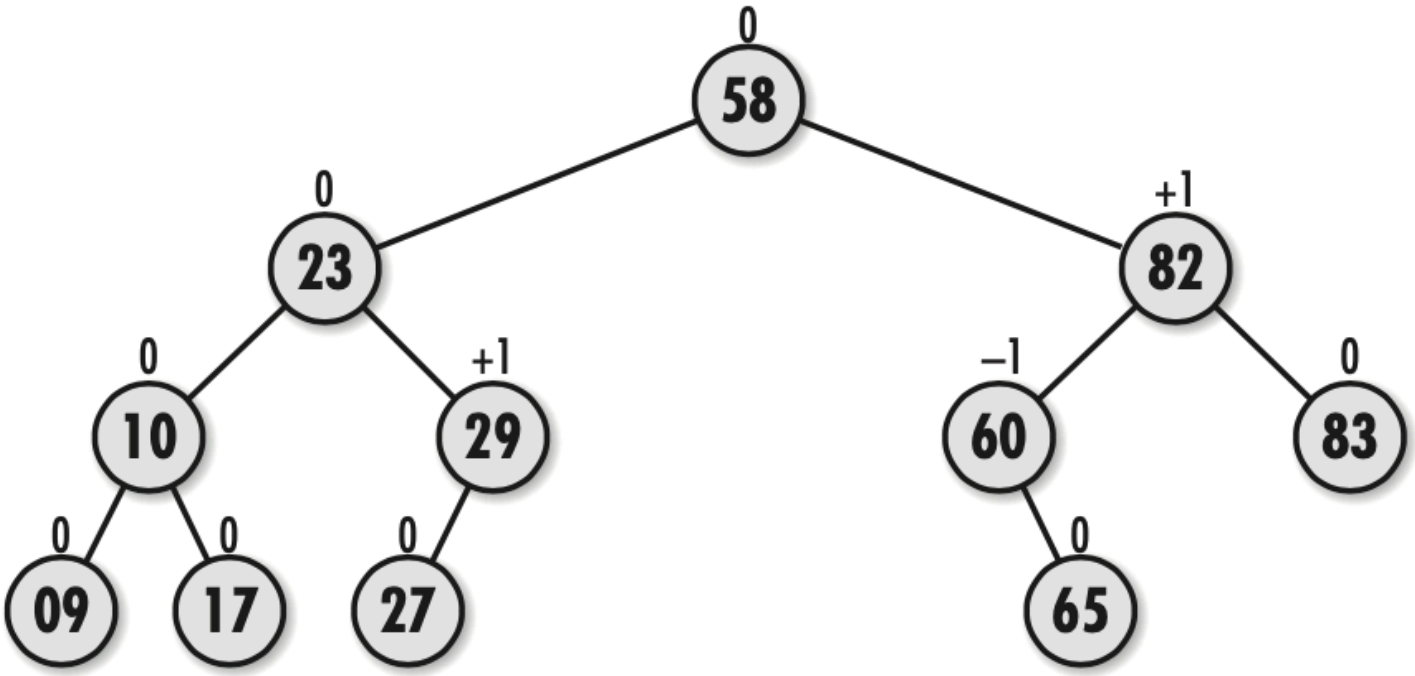
Définition d'un nœud

```
typedef struct integer_avlnode_ {
  int data;
  int hidden;
  int factor;
} integer_avlnode_t;
```

Définition d'un arbre

```
typedef generic_bitree_t integer_bistree_t;
```

Exemple : AVL avec indicateur d'équilibre



Implémentation des indicateur d'équilibre

```
#define          integer_AVL_LFT_HEAVY          1
#define          integer_AVL_BALANCED           0
#define          integer_AVL_RGT_HEAVY         -1
```

API des AVL – 1

API Création/Destruction

```
void integer_bistree_init(integer_bistree_t*);

void integer_bistree_destroy(integer_bistree_t*);
```

API Accesseurs / Prédicats

```
bool integer_bistree_lookup(integer_bistree_t*, int);

int integer_bistree_size(integer_bistree_t*);
```

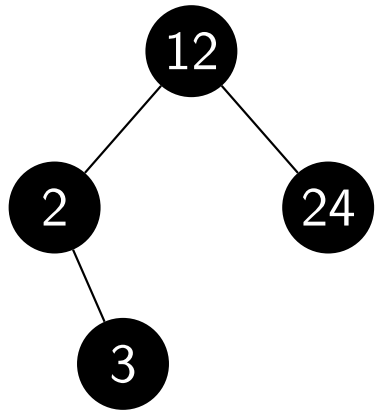
API Insertion/Suppression

```
int integer_bistree_insert(integer_bistree_t*, int);

int integer_bistree_remove(integer_bistree_t*, int);
```

Exemple : un petit AVL

Code source

```
int main (){

  int size = 4;
  int tab[] = {12, 2, 24, 3};

  integer_bistree_t avl;
  integer_bistree_init (&avl);

  for (int i = 0; i < size; i++) {
    integer_bistree_insert (&avl, tab[i]);
  }

  integer_bistree_destroy(&avl);

  return EXIT_SUCCESS;
}
```



Complexité des opérations sur les arbres binaires

Fonction	Complexité	Commentaire
Initialisation	$\mathcal{O}(1)$	Mise de la mémoire à zéro
Destruction	$\mathcal{O}(n)$	Parcourir tous les nœuds pour supprimer les éléments
Taille	$\mathcal{O}(1)$	
Insertion	$\mathcal{O}(\log(n))$	Il faut chercher où insérer
Suppression	$\mathcal{O}(\log(n))$	Il faut chercher où supprimer
Recherche	$\mathcal{O}(\log(n))$	L'équilibrage garantit une hauteur minimale

Remarque

La structure d'AVL est pensée pour stocker et faire des opérations de recherche

Remarque

L'implémentation des AVL dans la `libin103` utilise la technique nommée `lazy removal`. Elle ne supprime pas les données mais les cache (cf champs `hidden`).

Rotation

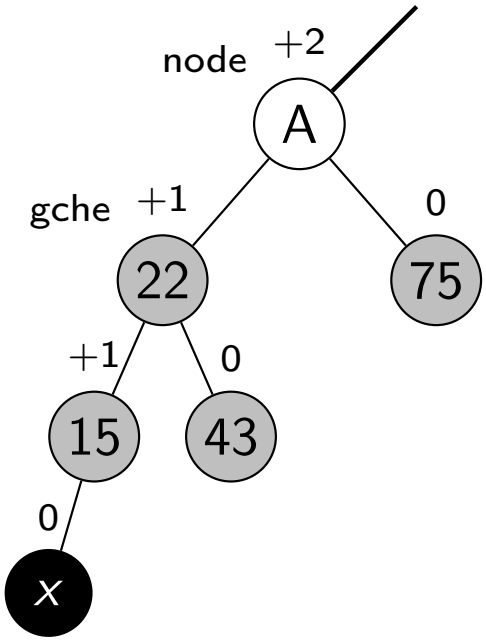
Pour maintenir un arbre équilibré les AVL effectuent des modifications locales de l'arbre après insertion : Rotation

Plusieurs cas sont envisagés :

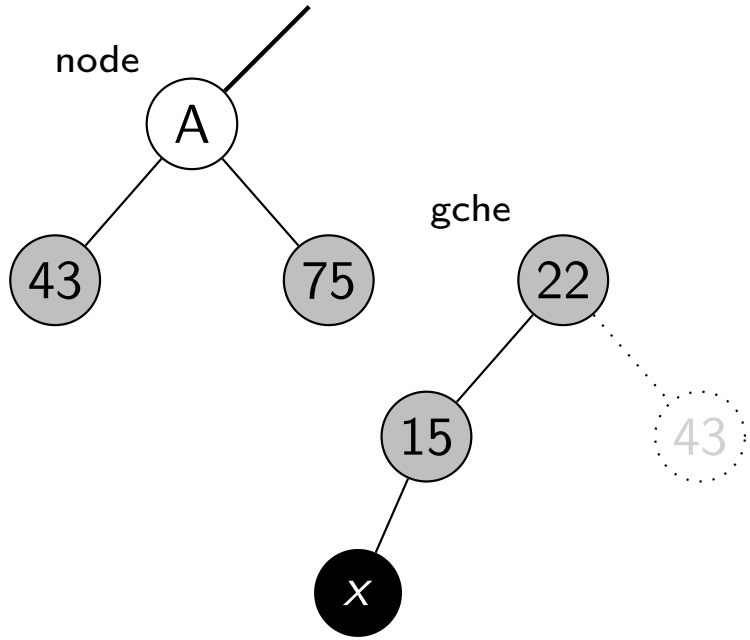
- Rotation LL (left-left)
  insertion d'un nouvel élément comme fils gauche du fils gauche
  $\Rightarrow$ Rotation à droite
- Rotation LR (left-right)
  insertion d'un nouvel élément comme fils droit du fils gauche
  $\Rightarrow$rotation à droite
- Rotation RR (right-right)
  insertion d'un nouvel élément comme fils droit du fils droit
  $\Rightarrow$rotation à gauche
- Roration RL (right-left)
  insertion d'un nouvel élément comme fils gauche du fils droit
  $\Rightarrow$rotation à gauche

Rotation : left-left (LL)

Étape 1 : insertion de x

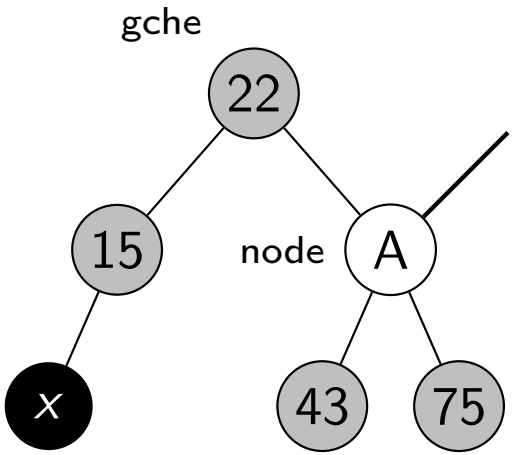


Étape 2 : `node->left = gche->right`

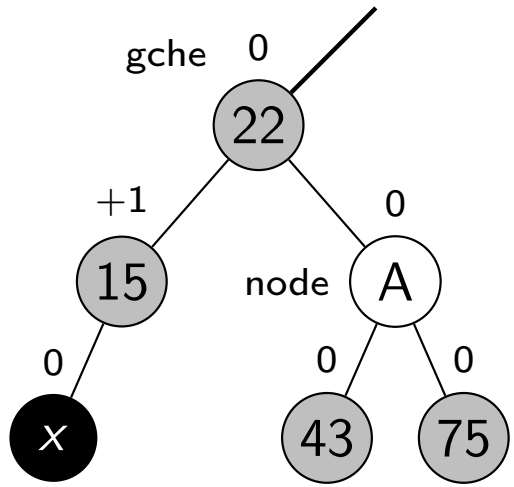


Rotation : left-left (LL)

Étape 3 : `gche->right = A`

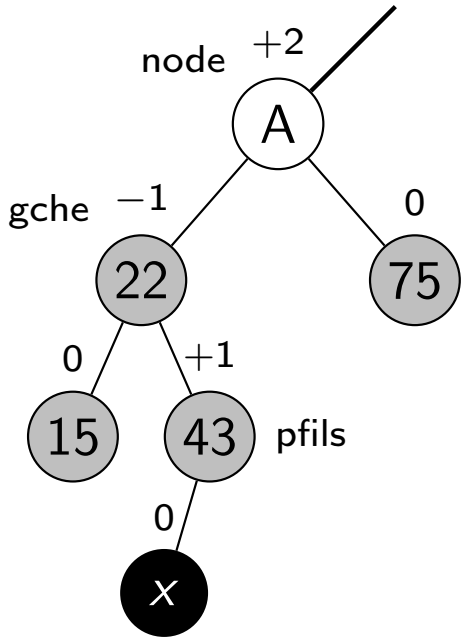


Étape 4 : Mise à jour du pointeur parent de A et des poids

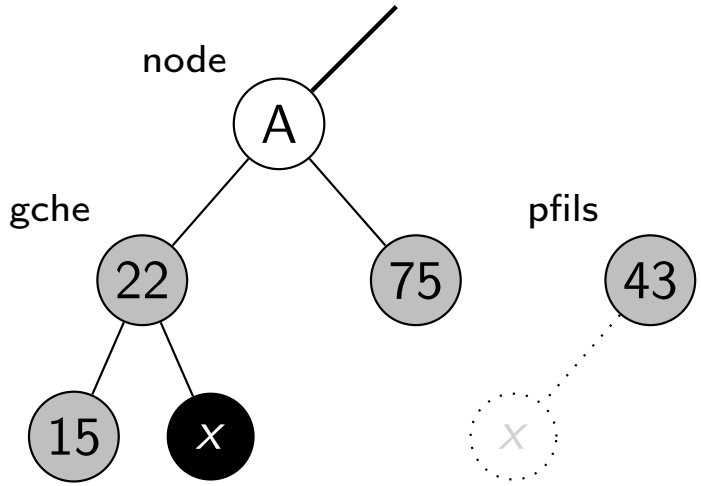


Rotation : left-right (LR)

Étape 1 : insertion de x

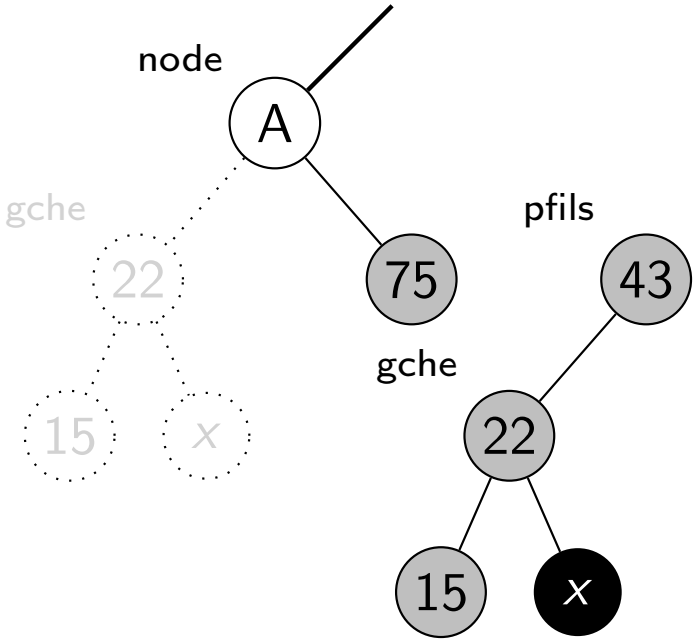


Étape 2 : `gche->right = pfils->left`

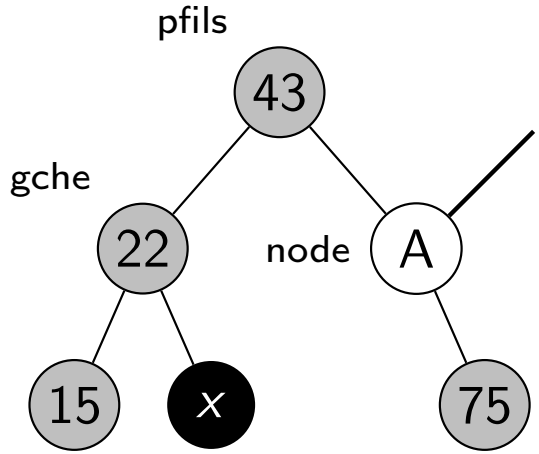


Rotation : left-right (LR)

Étape 3 : `pfils->left = gche`

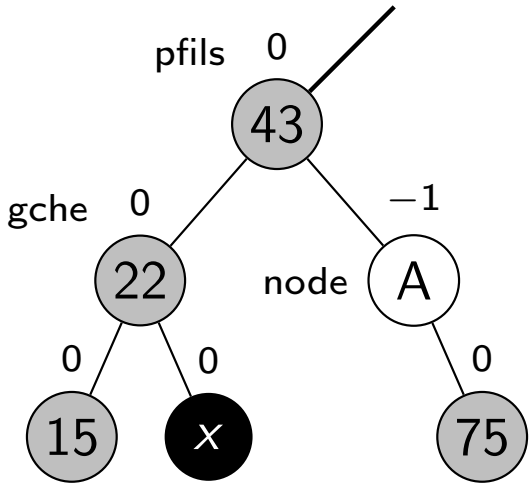


Étape 4 : `pfils->right = A`



Rotation : left-right (LR)

Étape 5 : Mise à jour du pointeur parent de A et des poids



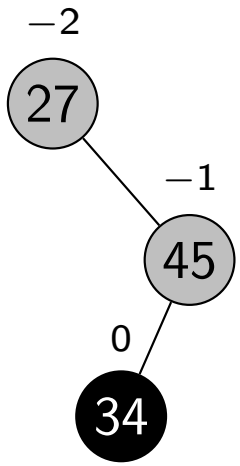
Remarque

Mise à jour des poids des nœuds `A`, `pfils` et `gche` dépend de la position d'insertion de `x`

- x est insérer comme fils droit de pfils (déséquilibre sur `A`, facteur -1)
- x est insérer comme pfils (équilibré)
- x est insérer comme fils gauche de pfils (déséquilibre sur `gche`, facteur $+1$)

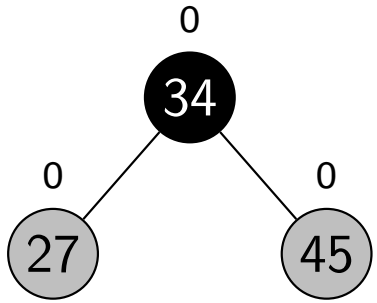
Exemple : construction d'un AVL – 1

Insertions consécutives de 27, 45, 34



Commentaire

L'insertion de 34 créé un déséquilibre qui déclanche une rotation RL.

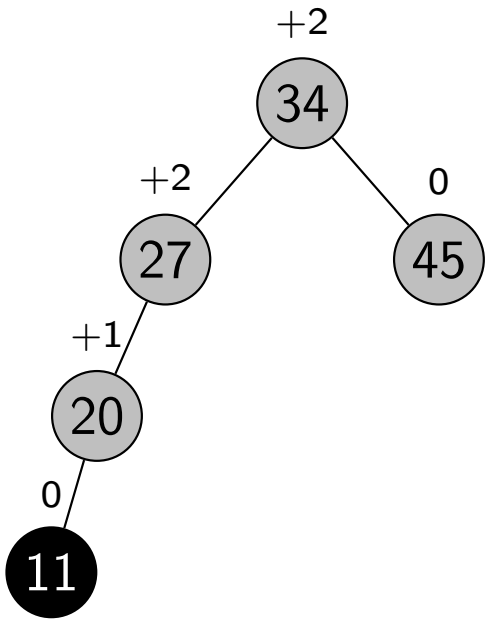


Commentaire

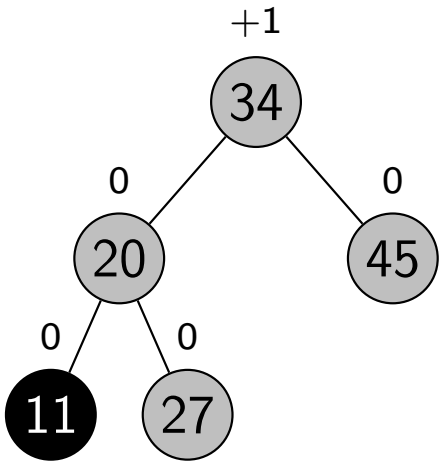
Après rotation RL

Exemple : construction d'un AVL – 2

Puis insertions consécutives de 20 et 11



‘



Commentaire

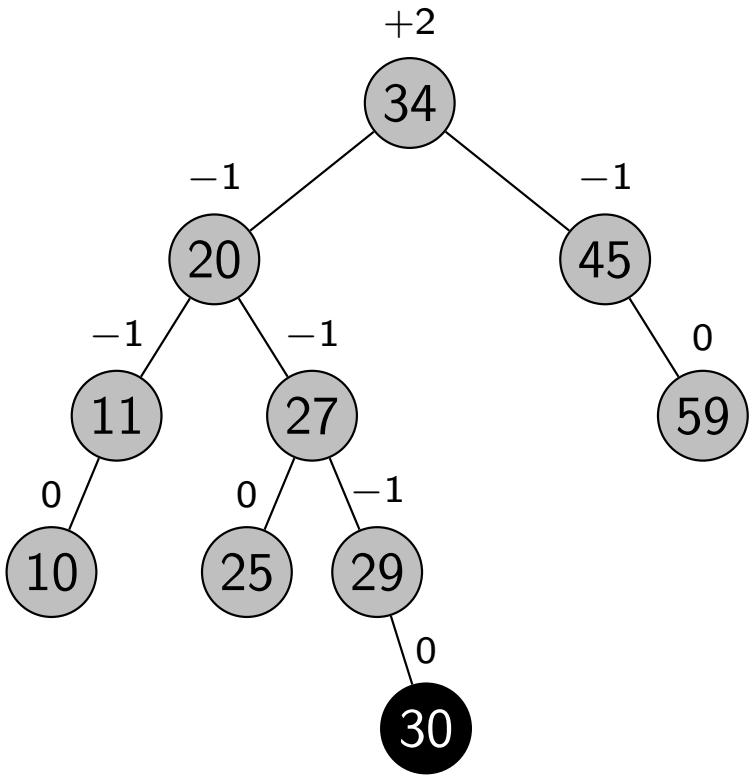
L'insertion de 11 créé un déséquilibre qui déclanche une rotation LL.

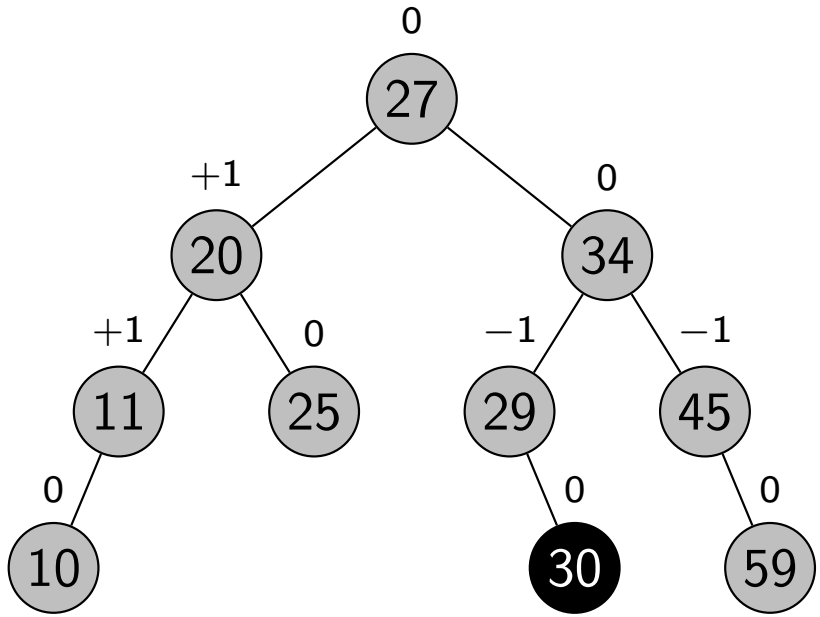
Commentaire

Après rotation LL

Exemple : construction d'un AVL – 3

Puis insertions consécutives de 59, 10, 25, 29 et 30





Commentaire

L'insertion de 30 créé un déséquilibre qui déclanche une rotation LR.

Commentaire

Après rotation LR

Une nouvelle organisation des données

Jusqu'à présent, des structures de données linéaires ont été étudiées :

- tableau : taille fixe, accès direct
- liste chainée : taille variable, accès indirect
- pile : comportement LIFO
- file : comportement FIFO
- ensemble : unicité des éléments
- arbres binaires de recherche (équilibrés)

Les arbres sont une structure de données associées à une relation d'ordre mais pas que

Tas

Motivation pour une nouvelle structure de données

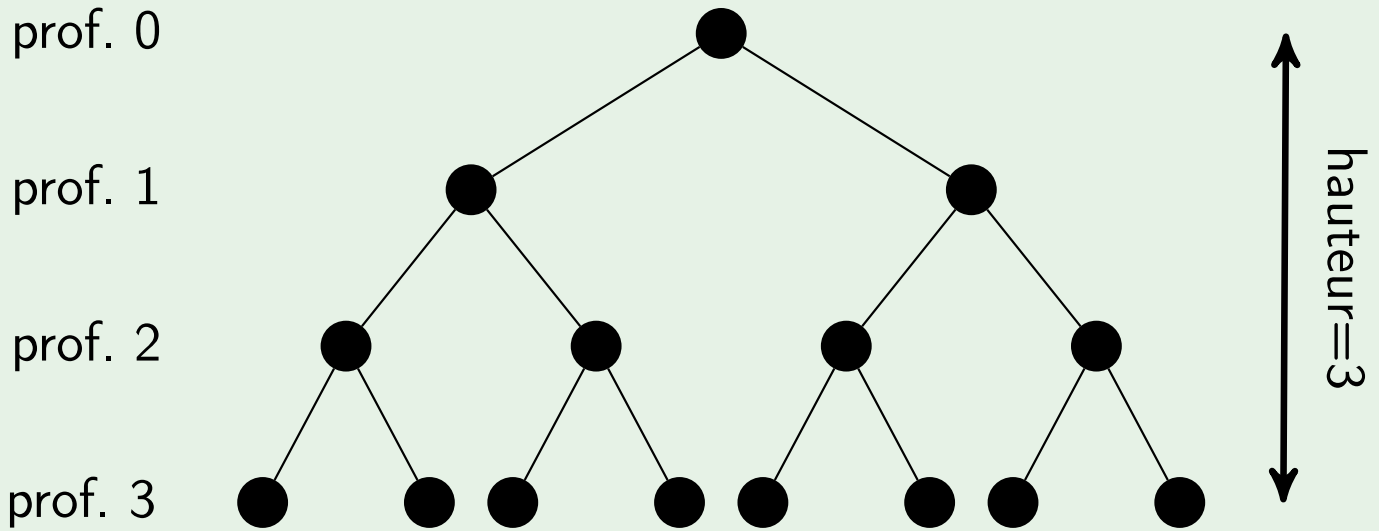
- Les files qui se comportent comme des files d'attente. On ne modifie pas l'ordre interne des éléments.
- Une file de priorité est une structure dynamique
  Fonctionnement comme une salle d'attente aux urgences : à chaque personne est attribuée une priorité de passage en fonction de la gravité de ses blessures et le traitement des patients est réalisé dans l'ordre des priorités avec potentiellement des préemptions suivant les nouvelles arrivées.
- Mise en œuvre par une structure de données tas
- Applications
  - Dans les systèmes d'exploitation, choix de l'ordre d'exécution des tâches
  - Dans les systèmes de correction orthographique, choix des propositions de correction (en fonction de distance comme celle de Levenshtein)
  - Tri de données
  - Dans les algorithmes de graphes : cf prochains cours

Arbres : encore du vocabulaire

Arbre k-aire complet

Un arbre dont toutes les feuilles sont à la même profondeur et tous les nœuds internes sont de degré k.

Arbre binaire complet



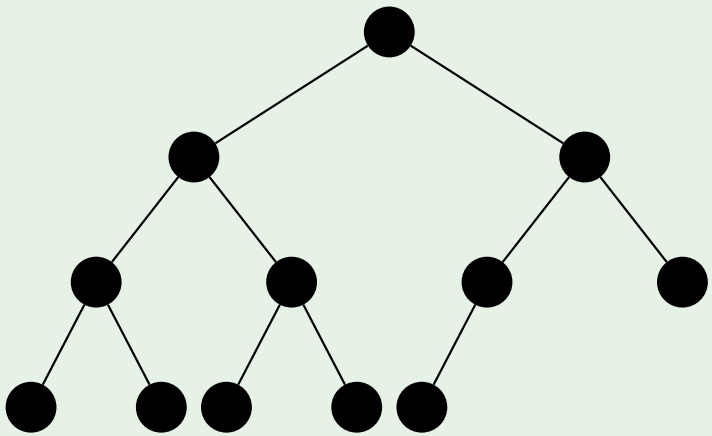
Remarque un arbre k-aire complet de hauteur h a k^h feuilles ; un arbre k-aire complet avec n feuilles a une hauteur $\log_k(n)$ (d'où l'équilibrage dans les AVL).

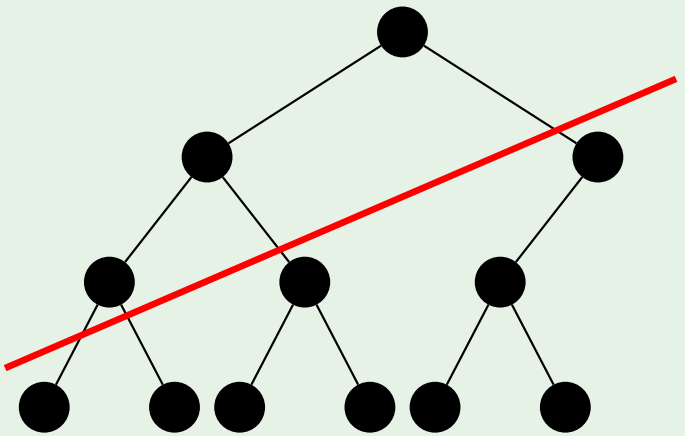
Arbres : encore du vocabulaire

Arbre k-aire quasi-complet ou tassé

Un arbre dont tous les niveaux sont remplis sauf éventuellement le dernier niveau qui est rempli sur la gauche.

Arbre binaire quasi-complet



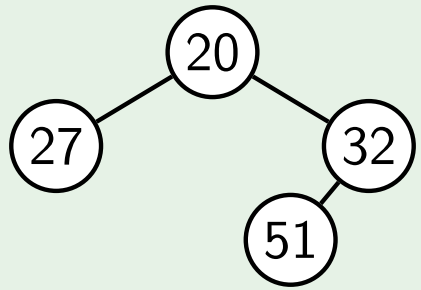


Arbres : encore du vocabulaire

Arbre binaire partiellement ordonné ou arbre tournoi

Suposons un ensemble de clefs muni d'un ordre total, un arbre binaire partiellement ordonné est tel que chaque nœud n a une valeur inférieure (ou supérieure) à ses enfants.

Exemple



Rappel : une relation binaire $\mathcal{R}$ sur un ensemble S est un ordre si elle est

- réflexive : $\forall x \in S,\ x \mathcal{R} x$
- antisymétrique $\forall x, y \in S,\ x \mathcal{R} y \wedge y \mathcal{R} x \Rightarrow x = y$
- transitive $\forall x, y, z \in S,\ x \mathcal{R} y \wedge y \mathcal{R} z \Rightarrow x \mathcal{R} z$

Un ordre est total si en plus $\forall x, y \in S,\ x \mathcal{R} y \vee y \mathcal{R} x$

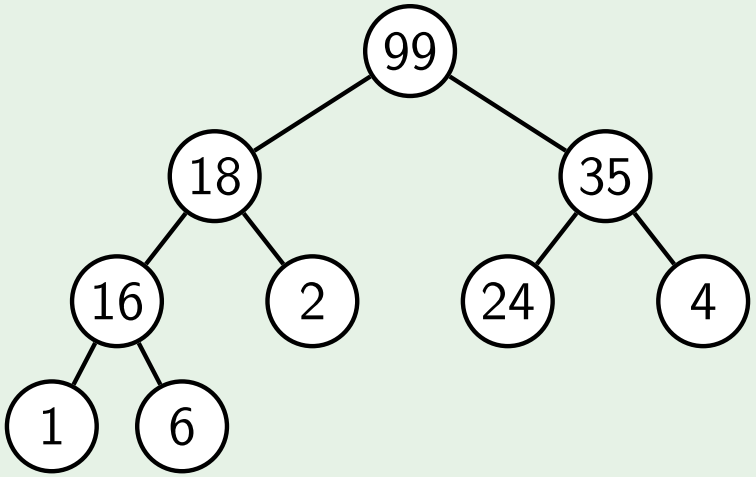
Structure de données Tas

Tas

Un tas est un arbre binaire quasi-complet partiellement ordonné. On parle de

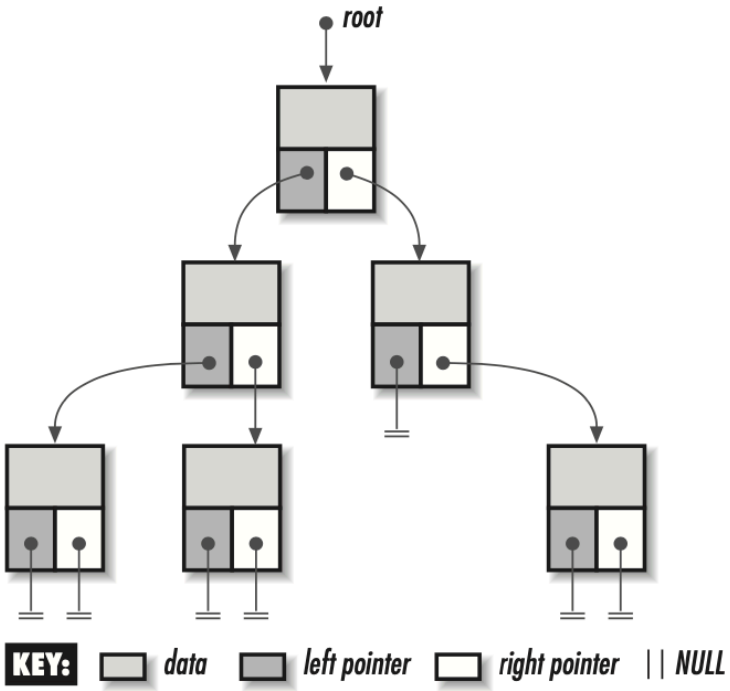
- Tas min quand la racine est la plus petite valeur de l'arbre
- Tas max quand la racine est la plus grande valeur de l'arbre

Exemple : tas max



Rappel implémentation par pointeurs des arbres

En langage C, les pointeurs sont utilisés pour faire un chaînage [3]



Définition d'un nœud

```
typedef struct integer_bitreenode_ {
  int data;
  struct integer_bitreenode_ *left;
  struct integer_bitreenode_ *right;
} integer_bitreenode_t;
```

Définition d'un arbre

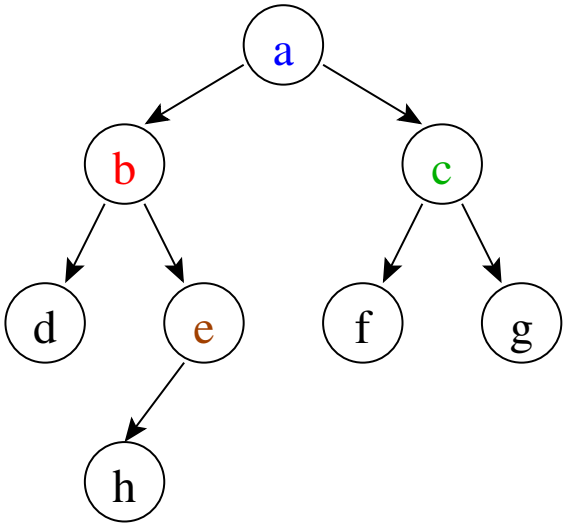
```
typedef struct integer_bitree_ {
  int size;
  integer_bitreenode_t *root;
} integer_bitree_t;
```

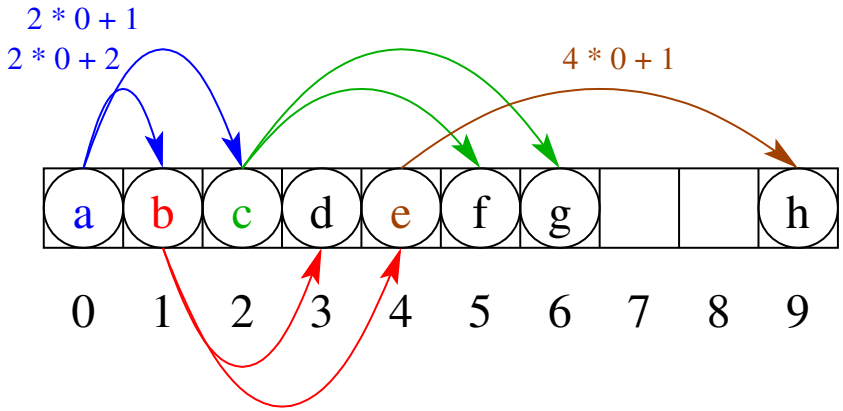
3. Crédit image : O'Reilly 'Mastering Algorithms with C'

Implémentation des arbres binaires par tableau

Avec un tableau :

- Un nœud a un indice i.
- Ses enfants sont aux indices $2i + 1$ et $2i + 2$.
- Son parent est à l'indice $\lfloor 0.5 \times (i - 1) \rfloor$ (avec $\lfloor \cdot \rfloor$ la partie entière inférieure).
- $\Rightarrow$ pas besoin de pointeurs.





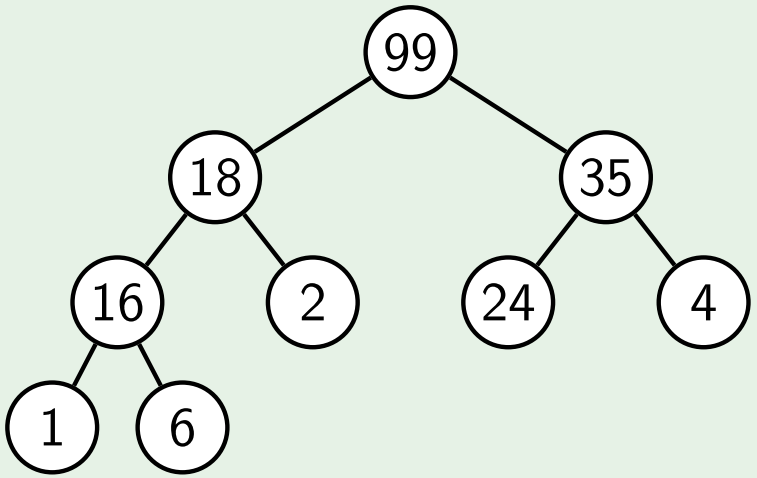
Remarque

Cette représentation est particulièrement adaptée pour la représentation des arbres binaires (quasi-) complets.

Représentation d'un tas avec un tableau

Exemple : tas max

0	1	2	3	4	5	6	7	8	9	10
99	18	35	16	2	24	4	1	6		



Remarque

La structure d'arbre quasi-complet permet d'éviter les espaces entre les éléments du tableau

Mise en œuvre dans `libin103`

Définition d'un tas

```
typedef enum integer_heap_type_ {
  integer_MAX_HEAP, integer_MIN_HEAP
} integer_heap_type_t;

typedef struct integer_heap_ {
  int size;
  integer_heap_type_t heap_type;
  int *tree;
} integer_heap_t;
```

Définitions de deux types

- Un type énuméré pour choisir un tas min ou tas max
- Une structure représentant un tas
  - taille
  - type de tas
  - tableau de valeurs respectant les proriétés d'arbre quasi-complet et partiellement ordonné

API des Tas

API Création/Destruction

```
void integer_heap_init(integer_heap_t*, integer_heap_type_t);

void integer_heap_destroy(integer_heap_t*);
```

API Accesseurs

```
int integer_heap_size(integer_heap_t*);
```

API Insertion/Suppression

```
int integer_heap_insert(integer_heap_t*, int);

int integer_heap_extract(integer_heap_t*, int*);
```

Complexité des opérations sur les tas

Fonction	Complexité	Commentaire
Initialisation	$\mathcal{O}(1)$	Mise à zéro des champs de la structure
Destruction	$\mathcal{O}(1)$ / $\mathcal{O}(n)$	Pour types primitifs / Parcourir tout le tableau pour supprimer les éléments
Taille	$\mathcal{O}(1)$	
Insertion	$\mathcal{O}(\log(n))$	Quelques permutations pour maintenir la propriété d'arbre partiellement ordonné
Extraction	$\mathcal{O}(\log(n))$	Quelques permutations pour maintenir la propriété d'arbre partiellement ordonné

Remarque

Les fonctions d'insertion et d'extraction modifient dynamiquement la taille du tableau de valeurs.

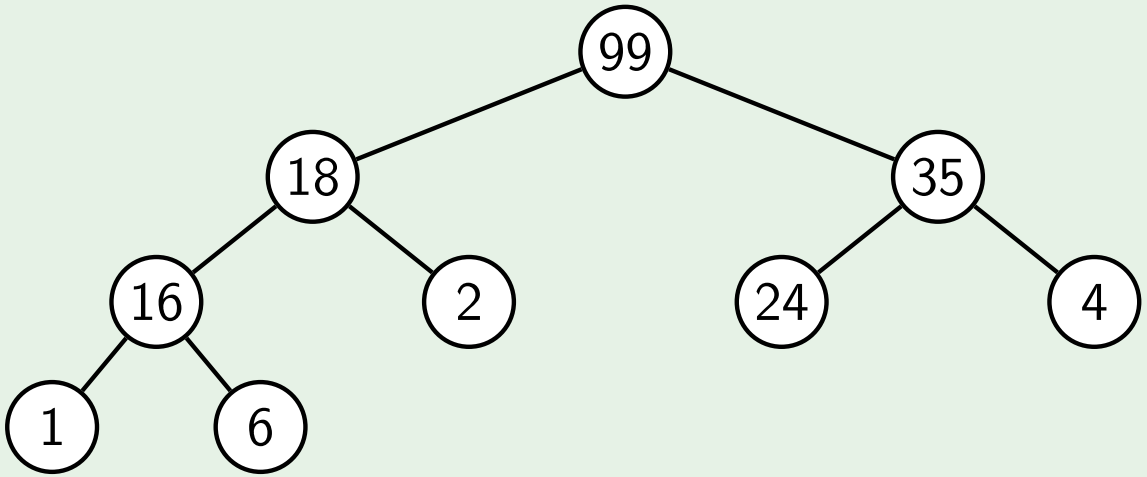
Insertion dans un tas max

Algorithme

- Placez la valeur dans la dernière case du tableau
- Tant que parent plus petit, permuter la valeur avec parent

Exemple : insertion de la valeur 22

0	1	2	3	4	5	6	7	8	9	10
99	18	35	16	2	24	4	1	6		



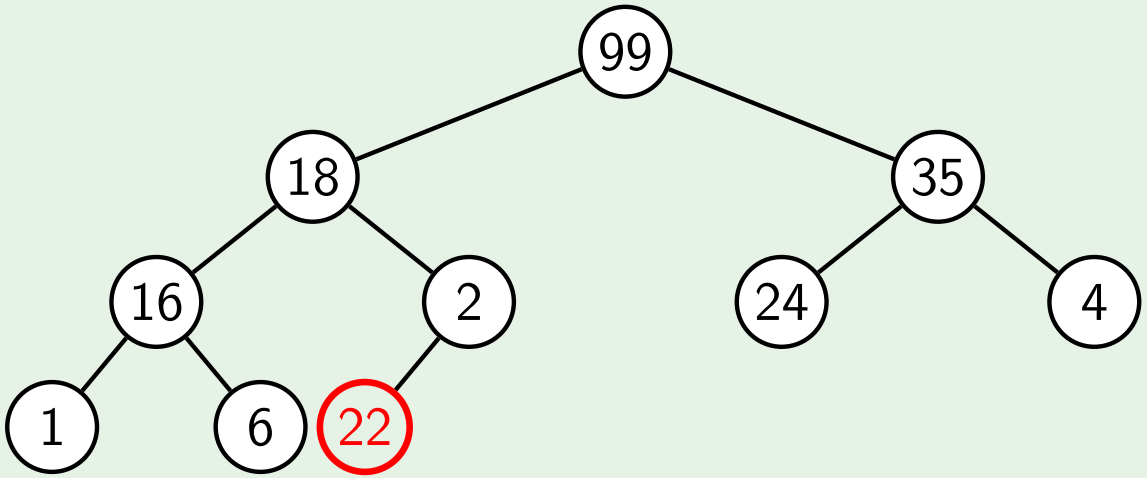
Insertion dans un tas max

Algorithme

- Placez la valeur dans la dernière case du tableau
- Tant que parent plus petit, permuter la valeur avec parent

Exemple : insertion de la valeur 22

0	1	2	3	4	5	6	7	8	9	10
99	18	35	16	2	24	4	1	6	22	



1. Insertion en fin de tableau

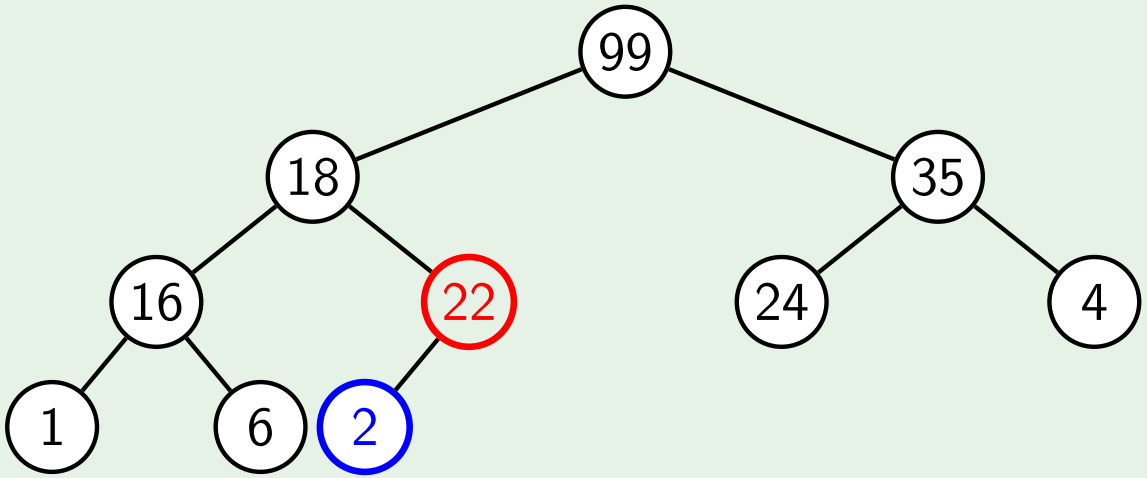
Insertion dans un tas max

Algorithme

- Placez la valeur dans la dernière case du tableau
- Tant que parent plus petit, permuter la valeur avec parent

Exemple : insertion de la valeur 22

0	1	2	3	4	5	6	7	8	9	10
99	18	35	16	22	24	4	1	6	2	



1. Insertion en fin de tableau
2. Permutation avec le parent car $2 < 22$

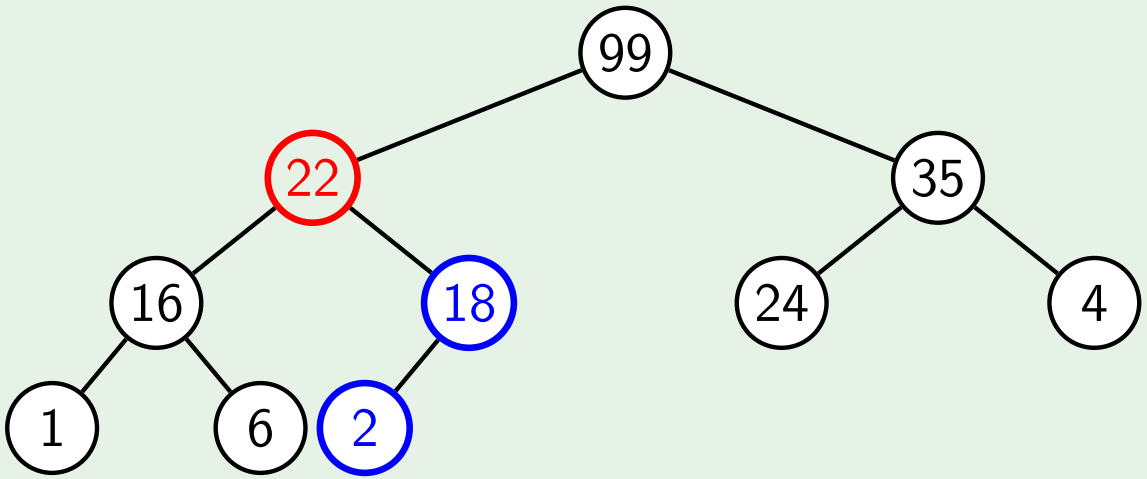
Insertion dans un tas max

Algorithme

- Placez la valeur dans la dernière case du tableau
- Tant que parent plus petit, permuter la valeur avec parent

Exemple : insertion de la valeur 22

0	1	2	3	4	5	6	7	8	9	10
99	22	35	16	18	24	4	1	6	2	



1. Insertion en fin de tableau
2. Permutation avec le parent car $2 < 22$
3. Permutation avec le parent car $18 < 22$
4. Fin

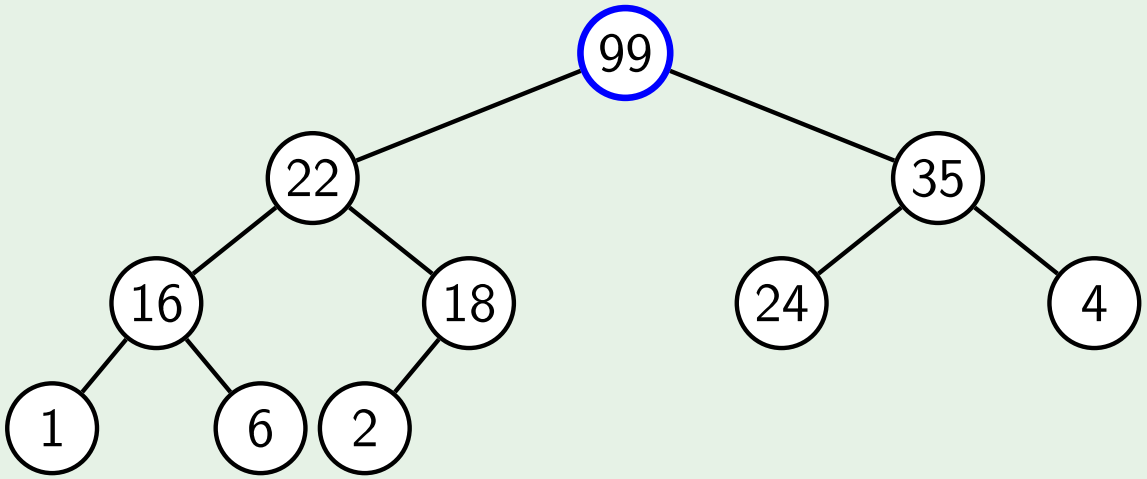
Extraction dans un tas max

Algorithme

- Copier la valeur du dernier nœud du tableau à la racine
- Faire redescendre la valeur de la racine en permutant avec le plus grand de ses enfants.
- Tant qu'un enfant a une plus grande valeur.

Exemple : extraction

0	1	2	3	4	5	6	7	8	9	10
99	22	35	16	18	24	4	1	6	2	



1. Extraction de la plus grande valeur

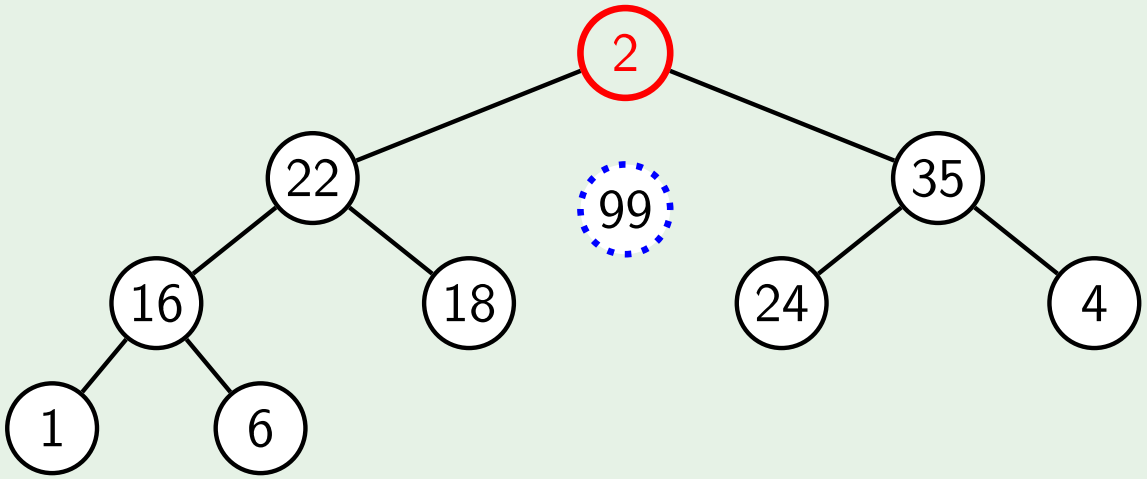
Extraction dans un tas max

Algorithme

- Copier la valeur du dernier nœud du tableau à la racine
- Faire redescendre la valeur de la racine en permutant avec le plus grand de ses enfants.
- Tant qu'un enfant a une plus grande valeur.

Exemple : extraction

0	1	2	3	4	5	6	7	8	9	10
2	22	35	16	18	24	4	1	6		



1. Extraction de la plus grande valeur
2. Permutation avec le dernier nœud

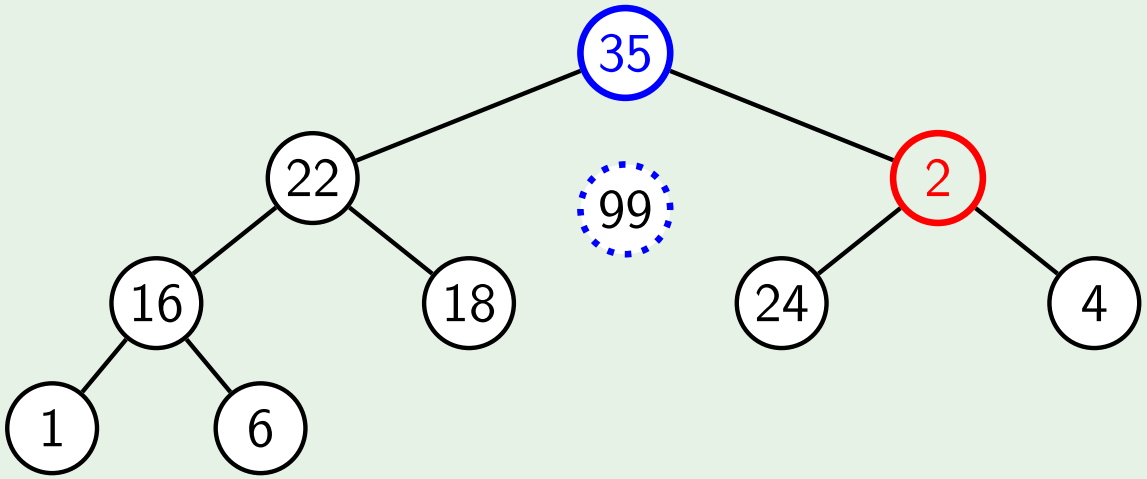
Extraction dans un tas max

Algorithme

- Copier la valeur du dernier nœud du tableau à la racine
- Faire redescendre la valeur de la racine en permutant avec le plus grand de ses enfants.
- Tant qu'un enfant a une plus grande valeur.

Exemple : extraction

0	1	2	3	4	5	6	7	8	9	10
35	22	2	16	18	24	4	1	6		



1. Extraction de la plus grande valeur
2. Permutation avec le dernier nœud
3. Permutation car $35 > 2$

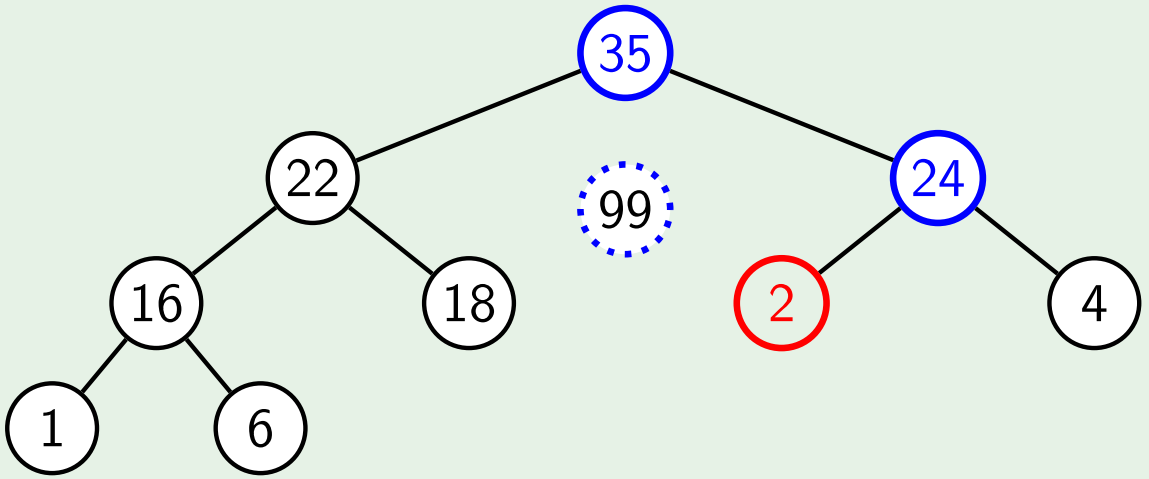
Extraction dans un tas max

Algorithme

- Copier la valeur du dernier nœud du tableau à la racine
- Faire redescendre la valeur de la racine en permutant avec le plus grand de ses enfants.
- Tant qu'un enfant a une plus grande valeur.

Exemple : extraction

0	1	2	3	4	5	6	7	8	9	10
35	22	24	16	18	2	4	1	6		



1. Extraction de la plus grande valeur
2. Permutation avec le dernier nœud
3. Permutation car $35 > 2$
4. Permutation car $24 > 2$
5. Fin

Exemple d'utilisation de la bibliothèque `libin103`

```c
#include <stdio.h>
#include <stdlib.h>
#include "heap.h"

int main (int argc, char* argv[]) {

  int size = 7;
  double tab[] = { 0.1, -2.0, 12.3, 3.14159, -1.34, 202.9, -2.67 };

  real_heap_t heap;
  real_heap_init (&heap, real_MIN_HEAP);

  for (int i = 0; i < size; i++) {
    real_heap_insert (&heap, tab[i]);
  }

  double min;
  real_heap_extract(&heap, &min);
  printf ("min␣=␣%f\n", min);

  real_heap_destroy (&heap);

  return EXIT_SUCCESS;
}
```

« Tassification » d'un tabeau

Une autre façon de construire un tas est de « tasser » un tableau existant.
Conséquence pas d'utilisation d'un second tableau pour le tas.

```
/* A appliquer sur tous les elements du tableau a partir de la fin */
Function heapify_max_from_index (array, index) :
    left = 2 * index + 1;
    right = 2 * index + 2;
    largest = index;
    if left < size(array) and  tab[left] > tab[largest] then
        largest = left;
    end
    if right < size(array) and  tab[rigth] > tab[largest] then
        largest = right;
    end
    if i != largest then
        swap (array[i], array[largest]);
        heapify_max_from_index (array, largest);
    end
```

Ensembles disjoints

Motivation pour une nouvelle structure de données

- La structure de données `Union-Find` (ou ensemble disjoint) est utilisée pour représenter des partitions d'un ensemble (d'entiers).

- La structure de données a 2 opérations :
  - `find(u)` : donne le sous-ensemble auquel appartient l'élément u. Un sous-ensemble est représenté par un témoin (un de ses éléments)
  - `union(u,v)` fusionne les deux sous-ensembles représentés par u et v.

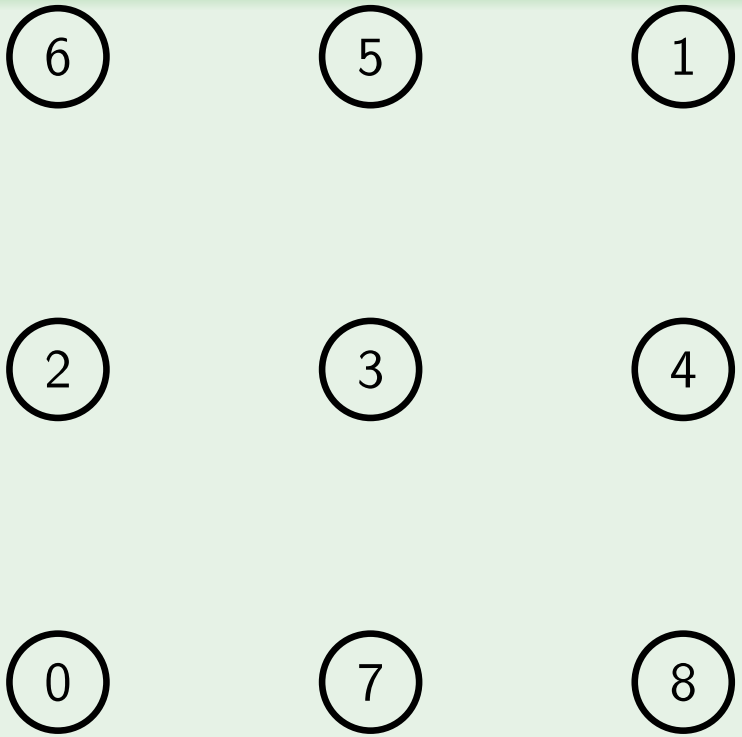
De nombreuses applications, cf plus loin

Fonctionnement d'un union-find

Point de départ :

- l'ensemble $S = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$
- représenté par 9 sous-ensembles singletons

Exemple

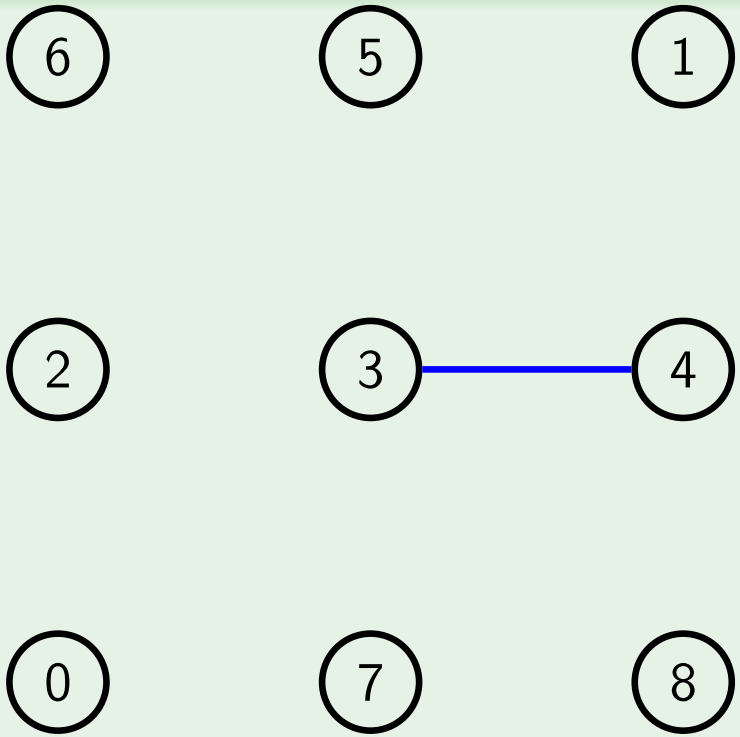


Fonctionnement d'un union-find

Point de départ :

- l'ensemble $S = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$
- représenté par 9 sous-ensembles singletons

Exemple



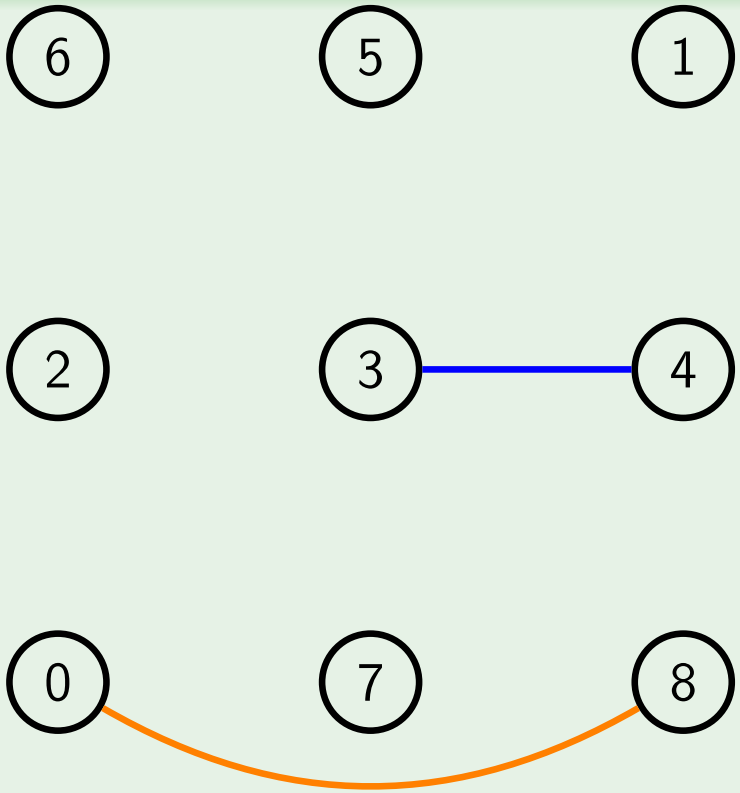
1. `union(3,4)`

Fonctionnement d'un union-find

Point de départ :

- l'ensemble $S = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$
- représenté par 9 sous-ensembles singletons

Exemple



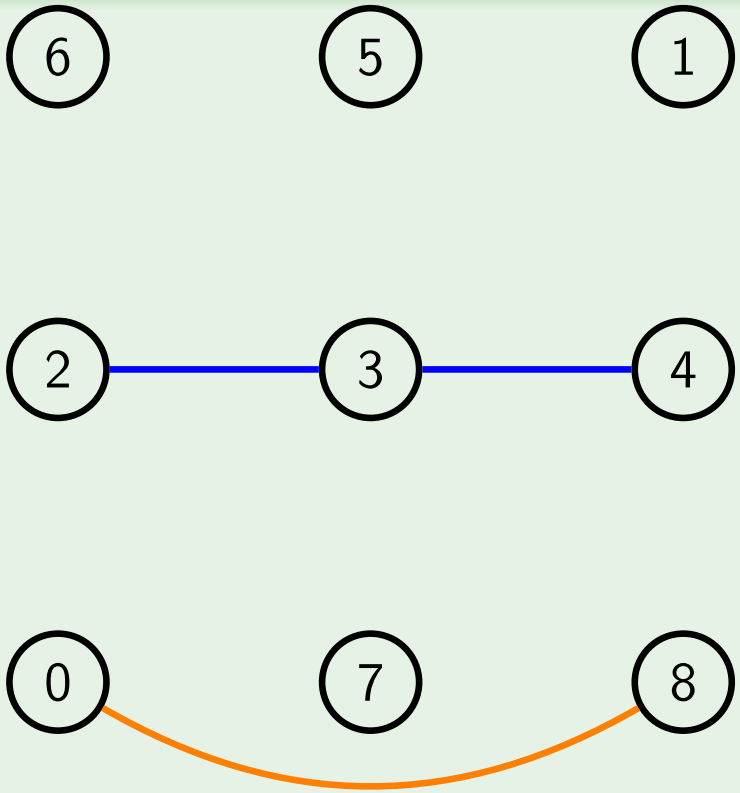
1. `union(3,4)`
2. `union(8,0)`

Fonctionnement d'un union-find

Point de départ :

- l'ensemble $S = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$
- représenté par 9 sous-ensembles singletons

Exemple



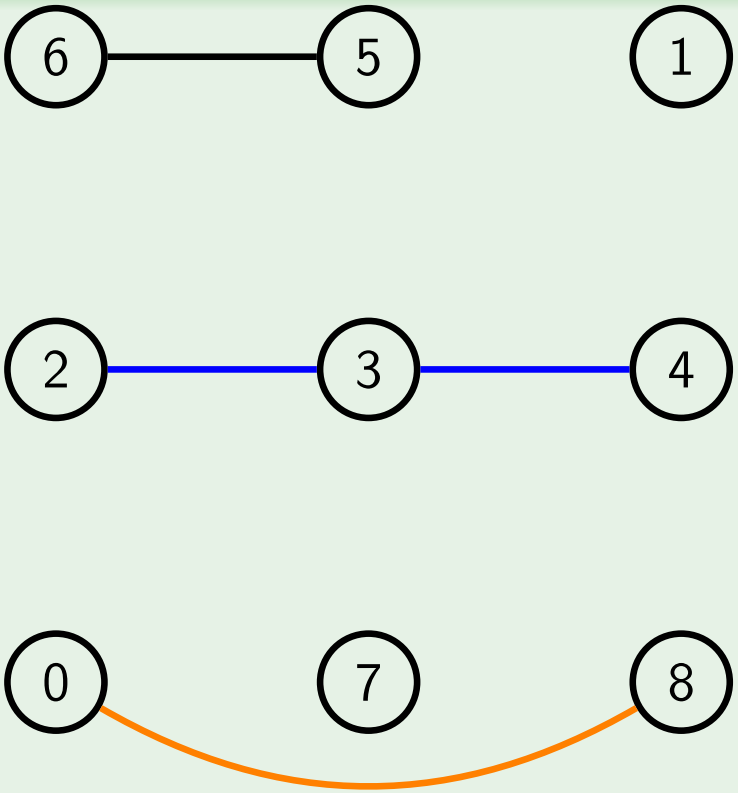
1. `union(3,4)`
2. `union(8,0)`
3. `union(2,3)`

Fonctionnement d'un union-find

Point de départ :

- l'ensemble $S = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$
- représenté par 9 sous-ensembles singletons

Exemple



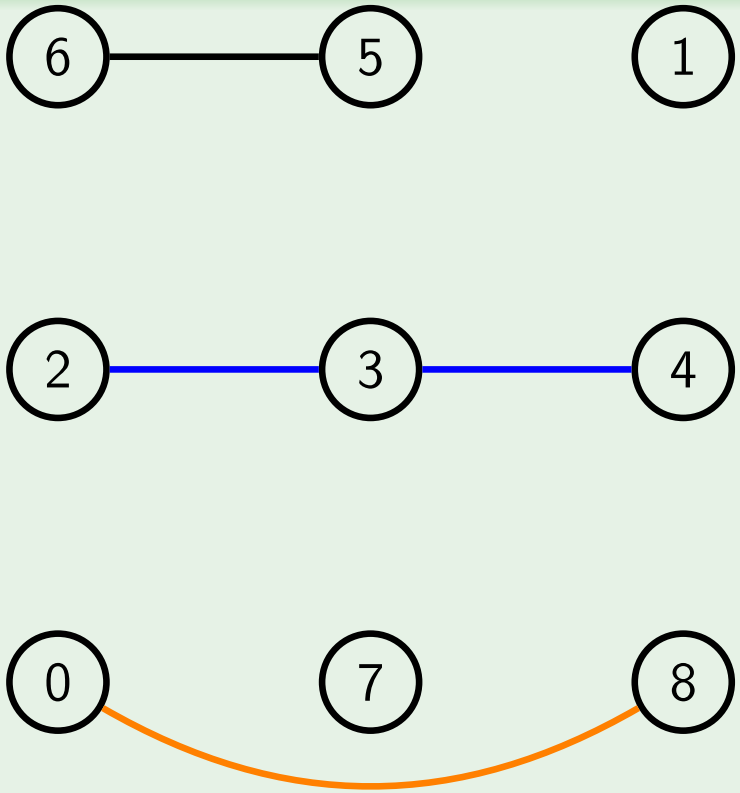
1. `union(3,4)`
2. `union(8,0)`
3. `union(2,3)`
4. `union(5,6)`

Fonctionnement d'un union-find

Point de départ :

- l'ensemble $S = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$
- représenté par 9 sous-ensembles singletons

Exemple



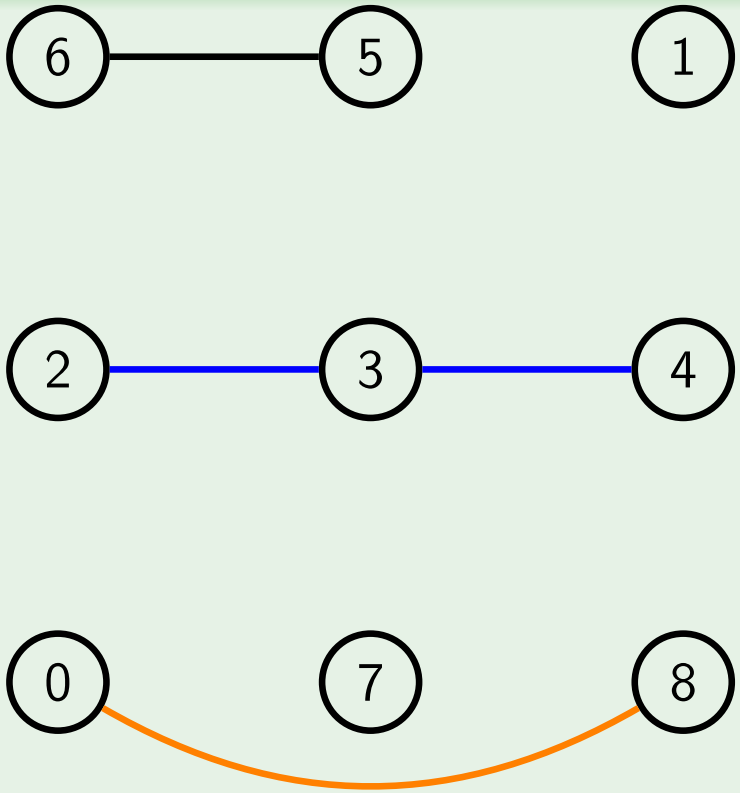
1. `union(3,4)`
2. `union(8,0)`
3. `union(2,3)`
4. `union(5,6)`
5. `are_connected(0,2)` $\Rightarrow$ non

Fonctionnement d'un union-find

Point de départ :

- l'ensemble $S = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$
- représenté par 9 sous-ensembles singletons

Exemple



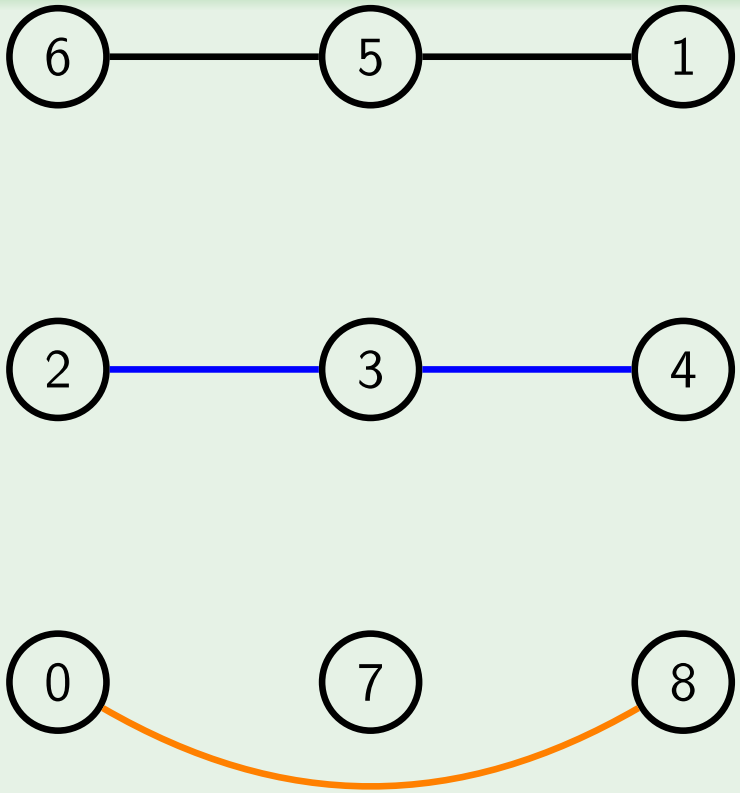
1. `union(3,4)`
2. `union(8,0)`
3. `union(2,3)`
4. `union(5,6)`
5. `are_connected(0,2)` $\Rightarrow$ non
6. `are_connected(2,4)` $\Rightarrow$ oui

Fonctionnement d'un union-find

Point de départ :

- l'ensemble $S = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$
- représenté par 9 sous-ensembles singletons

Exemple



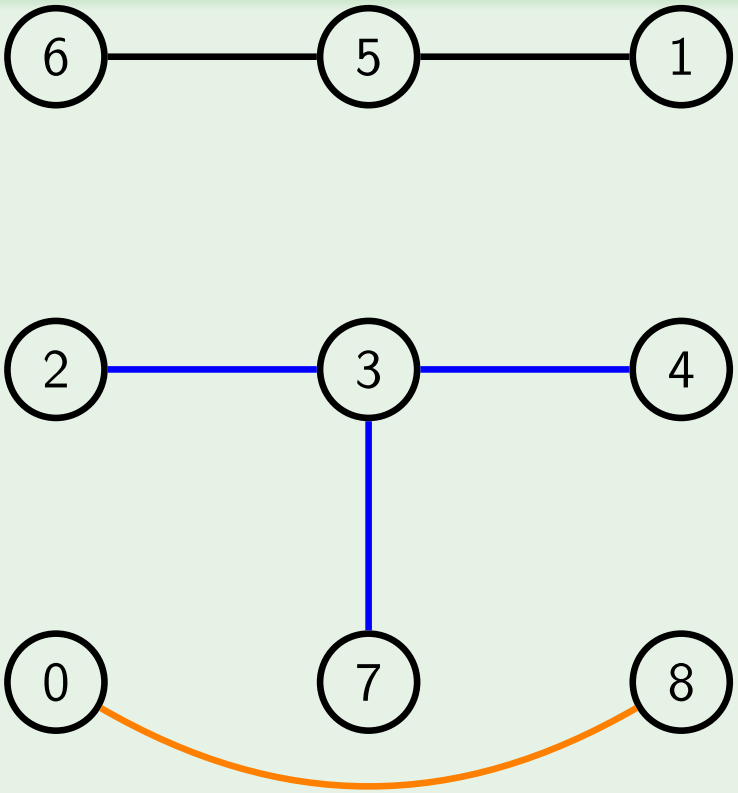
1. `union(3,4)`
2. `union(8,0)`
3. `union(2,3)`
4. `union(5,6)`
5. `are_connected(0,2)` $\Rightarrow$ non
6. `are_connected(2,4)` $\Rightarrow$ oui
7. `union(5,1)`

Fonctionnement d'un union-find

Point de départ :

- l'ensemble $S = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$
- représenté par 9 sous-ensembles singletons

Exemple



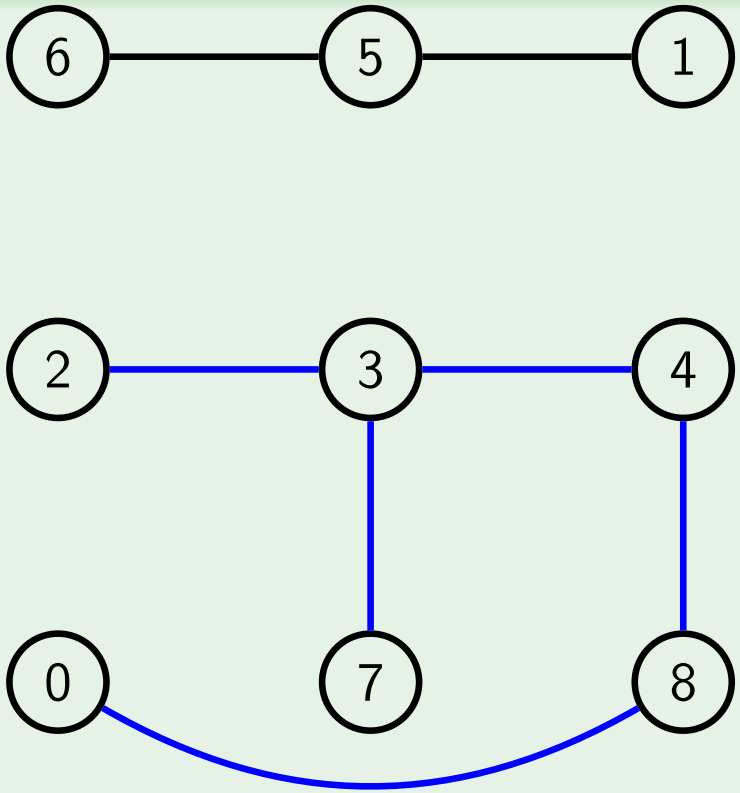
1. `union(3,4)`
2. `union(8,0)`
3. `union(2,3)`
4. `union(5,6)`
5. `are_connected(0,2)` $\Rightarrow$ non
6. `are_connected(2,4)` $\Rightarrow$ oui
7. `union(5,1)`
8. `union(7,3)`

Fonctionnement d'un union-find

Point de départ :

- l'ensemble $S = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$
- représenté par 9 sous-ensembles singletons

Exemple



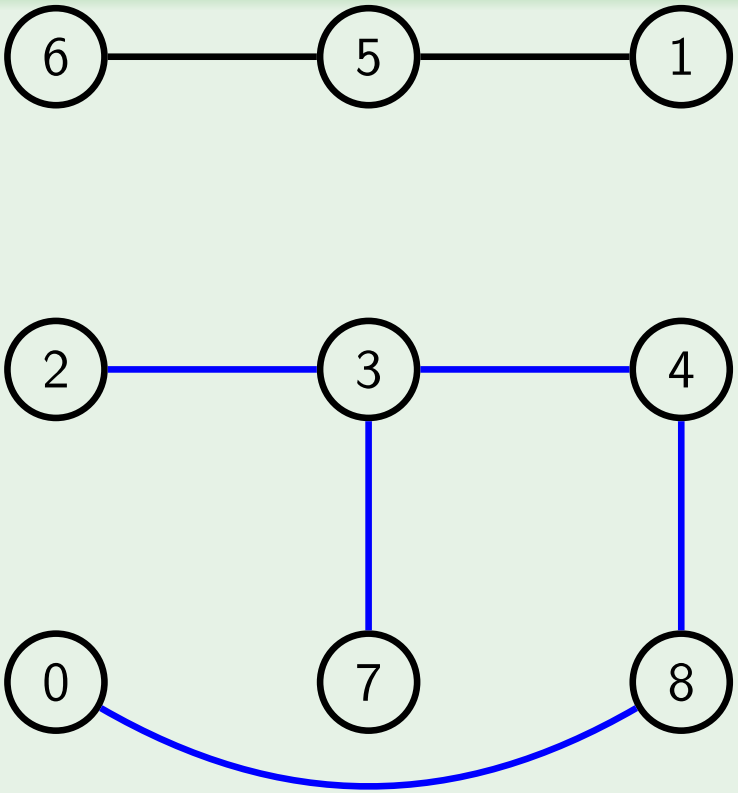
1. `union(3,4)`
2. `union(8,0)`
3. `union(2,3)`
4. `union(5,6)`
5. `are_connected(0,2)` $\Rightarrow$ non
6. `are_connected(2,4)` $\Rightarrow$ oui
7. `union(5,1)`
8. `union(7,3)`
9. `union(4,8)`

Fonctionnement d'un union-find

Point de départ :

- l'ensemble $S = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$
- représenté par 9 sous-ensembles singletons

Exemple



1. `union(3,4)`
2. `union(8,0)`
3. `union(2,3)`
4. `union(5,6)`
5. `are_connected(0,2)` $\Rightarrow$ non
6. `are_connected(2,4)` $\Rightarrow$ oui
7. `union(5,1)`
8. `union(7,3)`
9. `union(4,8)`
10. `are_connected(0,2)` $\Rightarrow$ oui
11. `are_connected(2,4)` $\Rightarrow$ oui

Relation d'équivalence et classe d'équivalence

Relation d'équivalence

Une relation $\mathcal{R}$ définie sur un ensemble S respectant les propriétés :

- Reflexivité : $\forall a \in S,\ a \mathcal{R} a$
- Symmétrie : $\forall a, b \in S,\ a \mathcal{R} b \Leftrightarrow b \mathcal{R} a$
- Transitivité : $\forall a, b, c \in S,\ a \mathcal{R} b \wedge b \mathcal{R} c \Rightarrow a \mathcal{R} c$

Classe d'équivalence

La classe d'équivalence de l'élément $a \in S$ est le sous-ensemble S_a de S tel que $S_a = \{x \in S : x \mathcal{R} a\}$.
Remarque l'ensemble des classes d'équivalence forme une partition.

La structure de données union-find permet de calculer dynamiquement les classes d'équivalence d'un ensemble avec

- l'opération `find(a)` qui donne la classe d'équivalence de a
- l'opération `union(x,y)` qui joint les classes d'équivalence de x et y en une seule classe.

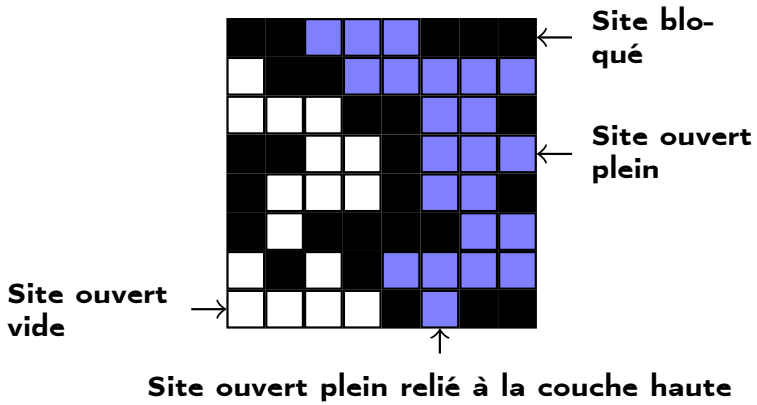
Applications des union-find

Percolation

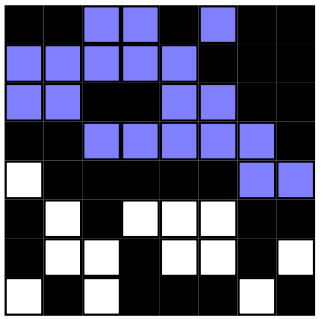
Désigne le passage d'un fluide à travers d'un milieu plus ou moins perméable.

Problèmes similaires : la conductivité d'un matériaux ou la facilité de communication dans les réseaux sociaux.

Percolation



Pas de percolation



Algorithme de Hoshen–Kopelman

- On considère une grille avec des cases notées occupées et libres
- Balayer des cases libres et numéroter les ensembles connexes.
- Percolation si les lignes de dessus et du dessous sont connectées.

Applications des union-find

- Segmentation d'images (similaire au problème de percolation)
- Utile dans les algorithmes de graphes (cf plus loin dans le cours)
- Procédure de décision en logique avec formules avec égalité et fonctions non interprétées
  - Permet de prouver l'insatisfiabilité par l'algorithme de fermeture congurente (*Congruence Closure Algorithm*)
  - P. ex., $f(a,b) = a \wedge f(f(a,b),b) \neq a$ est insatisfiable
    car a, $f(a,b)$ et $f(f(a,b),b)$ sont dans la même classe d'équivalence
- ...

Implémentation des union-find avec des forêts

Idées générales

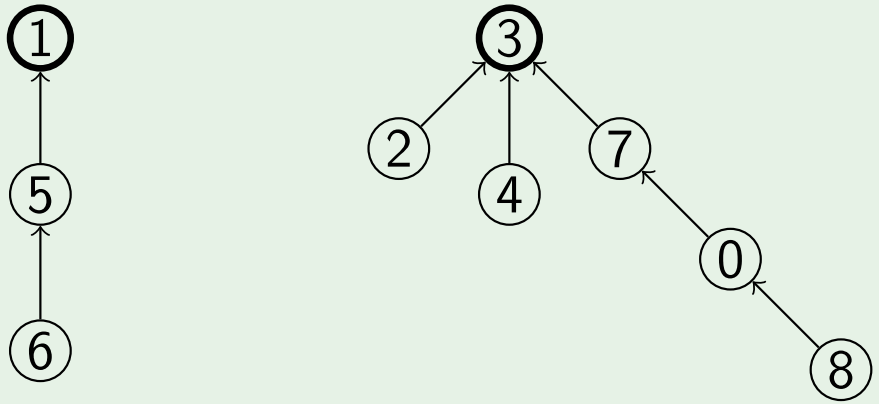
- Les éléments d'un ensemble sont stockés dans des nœuds d'arbres n-aires ;
- Il y a un arbre par classe d'équivalence d'où la notion de forêt ;
- La racine de l'arbre représente le témoin/représentant de la classe d'équivalence (valeur retournée par la fonction `find`) ;
- Les nœuds de l'arbre pointent vers les parents
  Attention c'est l'inverse des structures de données aborescentes vues jusqu'à présent !

Exemple

Une représentation des sous-ensembles

- $\{6, \underline{1}, 5\}$
- $\{0, 2, \underline{3}, 4, 7, 8\}$

pourrait être



Mise en œuvre dans `libin103`

Définition d'un union-find

```
typedef struct integer_uf_elm {
    struct integer_uf_elm *parent;
    int value;
    int depth;
} integer_uf_elm_t;

typedef struct integer_uf {
  integer_uf_elm_t** forest;
  int size;
  int components;
} integer_uf_t;
```

Remarque

Cette structure de données n'existe qu'en version avec des valeurs entières !

Un type élément d'un arbre (`integer_uf_elm_t`)

- avec un pointeur vers le parent
- une donnée
- une profondeur dans l'arbre

Un type pour la structure de donnée (`integer_uf_t`)

- un tableau de nœuds
- une taille du tableau, c'est-à-dire, la cardinalité de l'ensemble.
- un nombre de classes d'équivalence

API des union-find

API Création/Destruction

```
/* On donne le nombre max d'elements de l'ensemble */
void integer_uf_init (integer_uf_t*, int size);
void integer_uf_destroy(integer_uf_t*);
```

API Accesseurs et prédicats

```
int integer_uf_size (integer_uf_t*);
int integer_uf_components (integer_uf_t*);
bool integer_uf_are_connected (integer_uf_t*, int, int);
```

API Insertion/Suppression

```
int integer_uf_add_element(integer_uf_t*, int);
int integer_uf_find(integer_uf_t*, int, integer_uf_elm_t** result);
int integer_uf_union(integer_uf_t*, int, int);
```

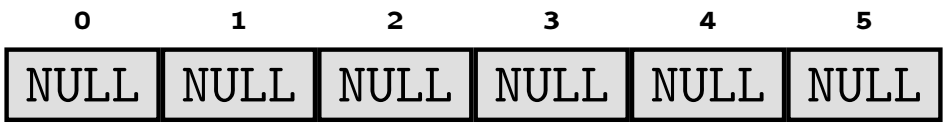
Complexité des opérations sur les union-find

Fonction	Complexité	Commentaire
Initialisation	$\mathcal{O}(1)$	Mise à zéro des champs de la structure (et allocation tableau)
Destruction	$\mathcal{O}(n)$	Parcourir tout le tableau pour supprimer les éléments
Taille / Composantes	$\mathcal{O}(1)$	
Union	$\mathcal{O}(\alpha(n))$	Repose sur la fonction `find`
Find	$\mathcal{O}(\alpha(n))$	Si utilisation d'heuristiques "path compression" et "union by rank"

Remarque

Les fonctions d'insertion et d'extraction modifient dynamiquement la taille du tableau de valeurs, gérée par la fonction `realloc`.

Illustration implémentation des union-find

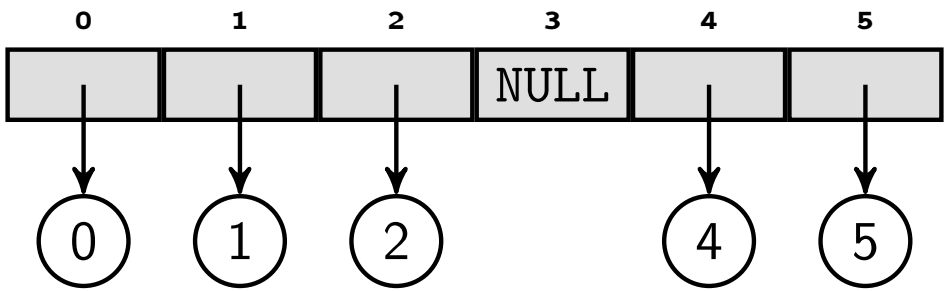


1. `init (&dset, 6);`

Remarque

On utilise des entiers pour avoir un accès direct aux éléments de l'ensemble avec les indices de tableau.

Illustration implémentation des union-find

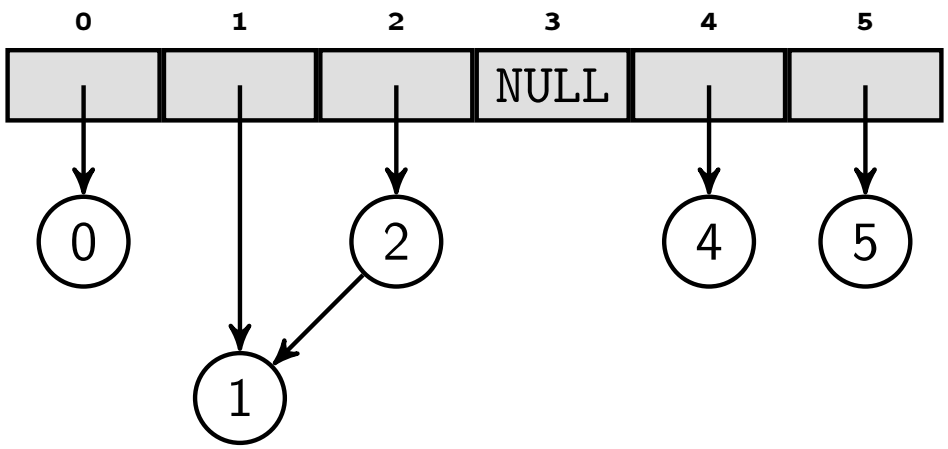


1. `init (&dset, 6);`
2. `add_element (&dset, 0);`
   (idem pour 1, 2, 4 et 5)

Remarque

On utilise des entiers pour avoir un accès direct aux éléments de l'ensemble avec les indices de tableau.

Illustration implémentation des union-find



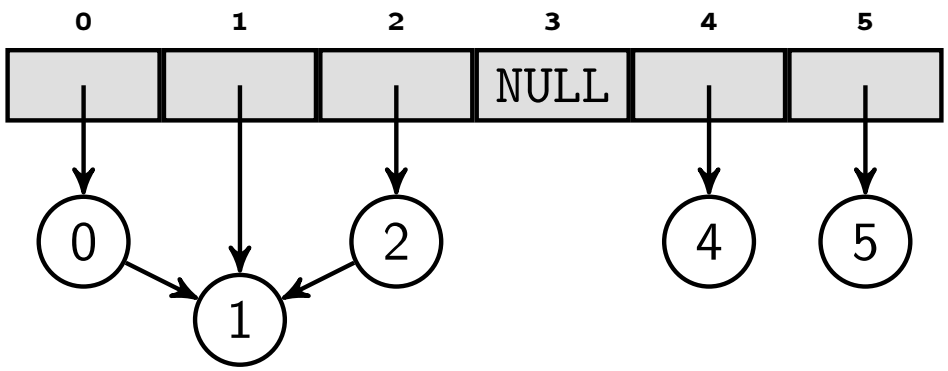
1. `init (&dset, 6);`
2. `add_element (&dset, 0);` (idem pour 1, 2, 4 et 5)
3. `union (&dset, 1, 2);`

Remarque

Opération d'union naïve a une complexité O(1) (mise à jour de pointeurs).

- Sans stratégie la hauteur de l'arbre peut grandir avec un impact sur la complexité de la fonction `find` $\in \mathcal{O}(n)$
- Stratégie *union by rank* on fait pointer le plus petit arbre vers le plus grand : `find` $\in \mathcal{O}(\log(n))$

Illustration implémentation des union-find



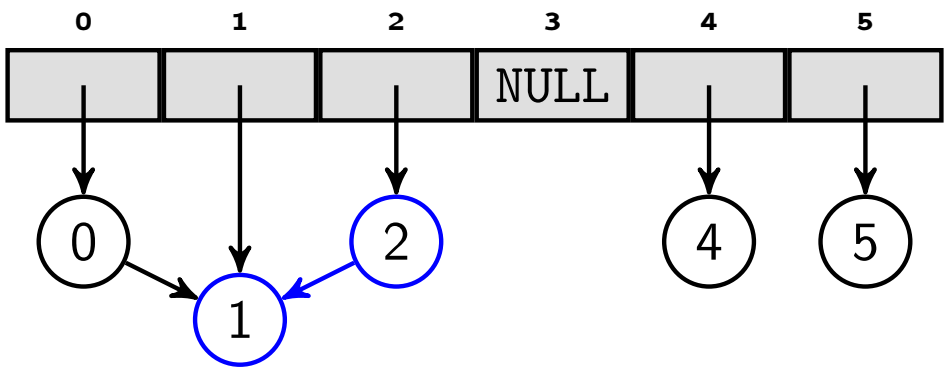
1. `init (&dset, 6);`
2. `add_element (&dset, 0);` (idem pour 1, 2, 4 et 5)
3. `union (&dset, 1, 2);`
4. `union (&dset, 0, 1);`

Remarque

Opération d'union naïve a une complexité O(1) (mise à jour de pointeurs).

- Sans stratégie la hauteur de l'arbre peut grandir avec un impact sur la complexité de la fonction `find` $\in \mathcal{O}(n)$
- Stratégie *union by rank* on fait pointer le plus petit arbre vers le plus grand : `find` $\in \mathcal{O}(\log(n))$

Illustration implémentation des union-find



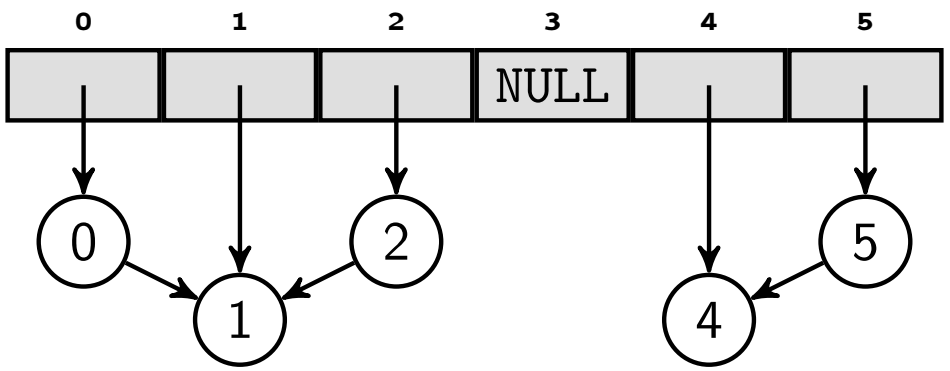
1. `init (&dset, 6);`
2. `add_element (&dset, 0);` (idem pour 1, 2, 4 et 5)
3. `union (&dset, 1, 2);`
4. `union (&dset, 0, 1);`
5. `find(&dset, 2);`

Remarque

Opération d'union naïve a une complexité O(1) (mise à jour de pointeurs).

- Sans stratégie la hauteur de l'arbre peut grandir avec un impact sur la complexité de la fonction `find` $\in \mathcal{O}(n)$
- Stratégie *union by rank* on fait pointer le plus petit arbre vers le plus grand : `find` $\in \mathcal{O}(\log(n))$

Illustration implémentation des union-find



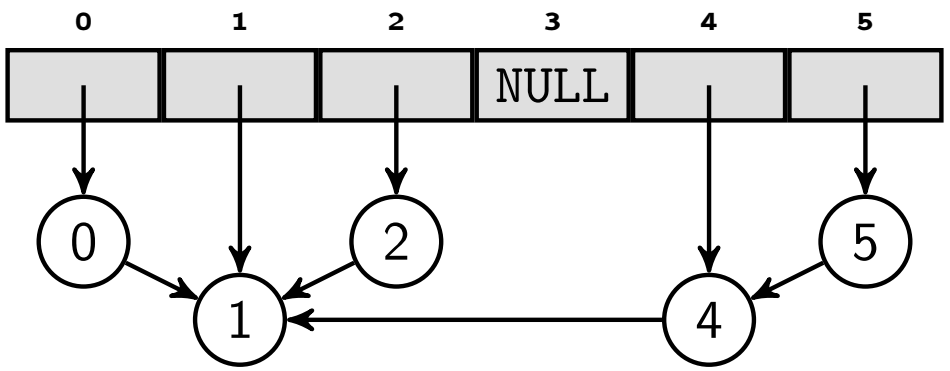
1. `init (&dset, 6);`
2. `add_element (&dset, 0);` (idem pour 1, 2, 4 et 5)
3. `union (&dset, 1, 2);`
4. `union (&dset, 0, 1);`
5. `find(&dset, 2);`
6. `union (&dset, 4, 5);`

Remarque

Opération d'union naïve a une complexité O(1) (mise à jour de pointeurs).

- Sans stratégie la hauteur de l'arbre peut grandir avec un impact sur la complexité de la fonction `find` $\in \mathcal{O}(n)$
- Stratégie *union by rank* on fait pointer le plus petit arbre vers le plus grand : `find` $\in \mathcal{O}(\log(n))$

Illustration implémentation des union-find



1. `init (&dset, 6);`
2. `add_element (&dset, 0);` (idem pour 1, 2, 4 et 5)
3. `union (&dset, 1, 2);`
4. `union (&dset, 0, 1);`
5. `find(&dset, 2);`
6. `union (&dset, 4, 5);`
7. `union (&dset, 2, 4);`

Remarque

Opération d'union naïve a une complexité O(1) (mise à jour de pointeurs).

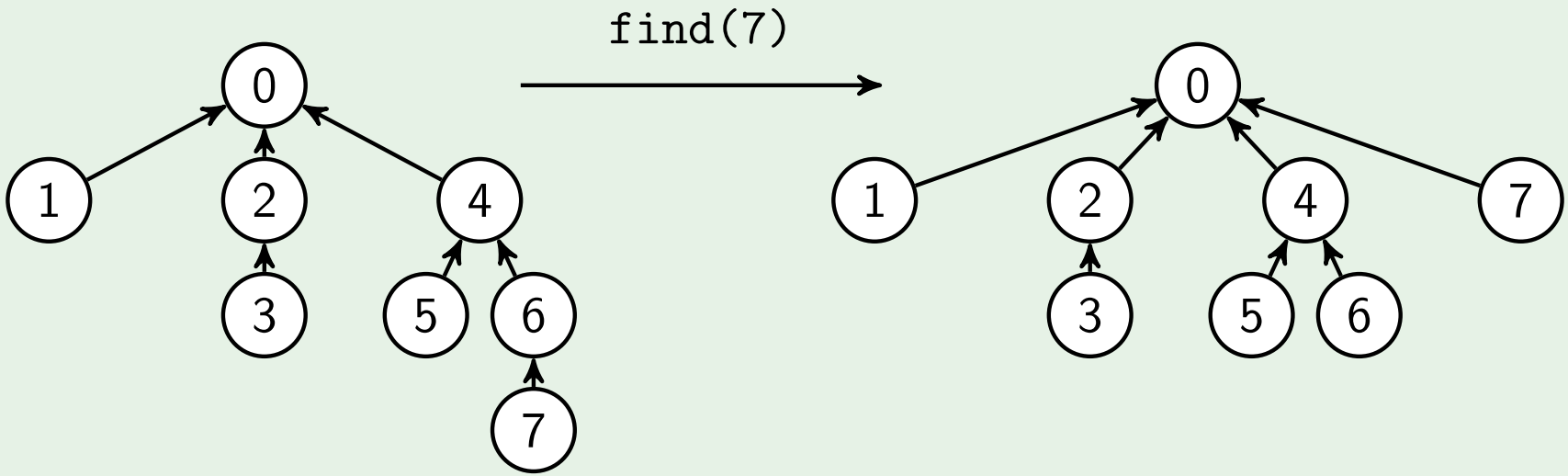
- Sans stratégie la hauteur de l'arbre peut grandir avec un impact sur la complexité de la fonction `find` $\in \mathcal{O}(n)$
- Stratégie *union by rank* on fait pointer le plus petit arbre vers le plus grand : `find` $\in \mathcal{O}(\log(n))$

Opérations d'union-find améliorées

Path compression

Lors de l'appel à `find` le nœud en argument est modifié pour pointer directement vers la racine.

Exemple du fonctionnement dans la bibliothèque `libin103`



Remarque

On peut encore améliorer la complexité en reliant tous les nœuds sur le chemin vers la racine dans l'appel à `find` à la racine.

Un mot sur la complexité (pour le fun)

La complexité amortie[4] de la fonction `find` est $\mathcal{O}(m\alpha(n))$ avec m le nombre d'opérations et n le nombre d'éléments dans l'ensemble.

Analyse de complexité amortie

Calcul de complexité sur l'exécution d'une séquence d'opérations sur une structure des données. (En général, analyse difficile)

Fonction d'Ackermann

$$A(0,n) = n+1$$
$$A(m+1,0) = A(m,1)$$
$$A(m+1,n+1) = A(m,A(m+1,n))$$

- A a une croissance très très rapide,
- Par exemple, $A(4,2)$ est un nombre avec 19729 chiffres équivalent à $2^{65536}-3$!

α représente la fonction inverse de la fonction d'Ackermann ($\alpha(n) = \min\{k : A(k,1) \geqslant n\}$) donc une fonction à croissance très très faible !

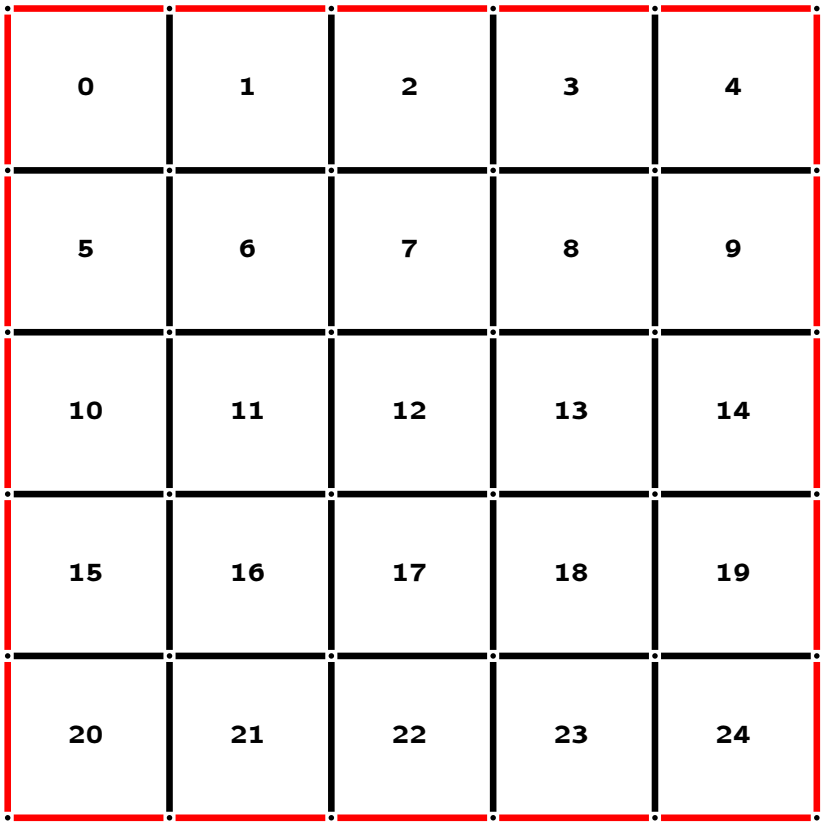
4. voir "Introduction to algorithms" de Thomas H. Cormen *et al.* Chap. 17 et 21

Exemple d'utilisation de la bibliothèque `libin103`

```c
#include <stdio.h>
#include <stdlib.h>
#include "integer_uf.h"

int main (){
  int size = 5; integer_uf_t uf;
  integer_uf_init (&uf, size);

  for (int i = 0; i < size; i++) {
    integer_uf_add_element (&uf, i);
  }

  integer_uf_union (&uf, 1, 3);
  integer_uf_union (&uf, 2, 4);
  integer_uf_union (&uf, 4, 1);

  if (integer_uf_are_connected (&uf, 3, 2)) {
    printf ("3␣et␣2␣sont␣dans␣la␣meme␣classe\n");
  }

  integer_uf_destroy (&uf);

  return EXIT_SUCCESS;
}
```

Application : génération de labyrinthe (voir TP)

- Un petit labyrinthe carré de 5 cases de côté ;
- Considérer toutes les cases de 0 à 24 comme singletons ;
- Liste des murs internes $S = \{(0,1),(0,5),(1,2),(1,6),(2,3),\dots\}$.
- M est la liste des murs à conserver durant la construction du labyrinthe.

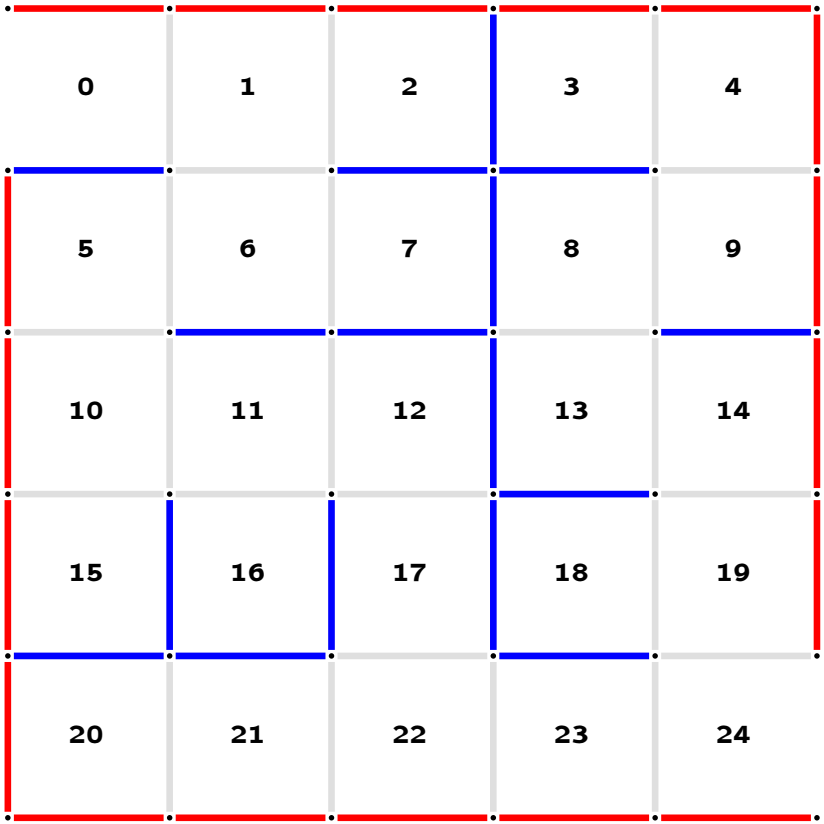


Algorithme

- Tant que toutes les cases ne sont pas dans le même ensemble :
  - prendre au hasard $(c_1, c_2) \in S$
  - si c_1 et c_2 sont dans le même ensemble, le mur (c_1, c_2) est conservé, mettre dans M
  - si c_1 et c_2 ne sont pas dans le même ensemble, `union(`c_1`,` c_2`)` et ne plus considérer (c_1, c_2)
- $M \cup S$ sont les murs du labyrinthe.
- Choisir une entrée et une sortie

Application : génération de labyrinthe (voir TP)

- Un petit labyrinthe carré de 5 cases de côté ;
- Considérer toutes les cases de 0 à 24 comme singletons ;
- Liste des murs internes $S = \{(0,1),(0,5),(1,2),(1,6),(2,3),\dots\}$.
- M est la liste des murs à conserver durant la construction du labyrinthe.



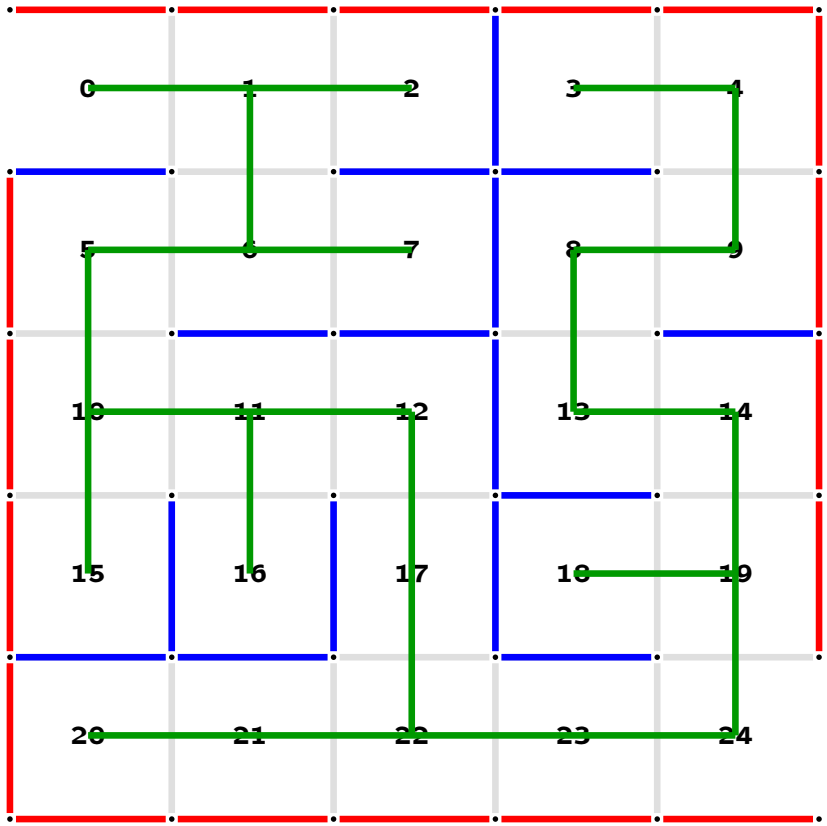
Algorithme

- Tant que toutes les cases ne sont pas dans le même ensemble :
  - prendre au hasard $(c_1, c_2) \in S$
  - si c_1 et c_2 sont dans le même ensemble, le mur (c_1, c_2) est conservé, mettre dans M
  - si c_1 et c_2 ne sont pas dans le même ensemble, `union(`c_1`,` c_2`)` et ne plus considérer (c_1, c_2)
- $M \cup S$ sont les murs du labyrinthe.
- Choisir une entrée et une sortie

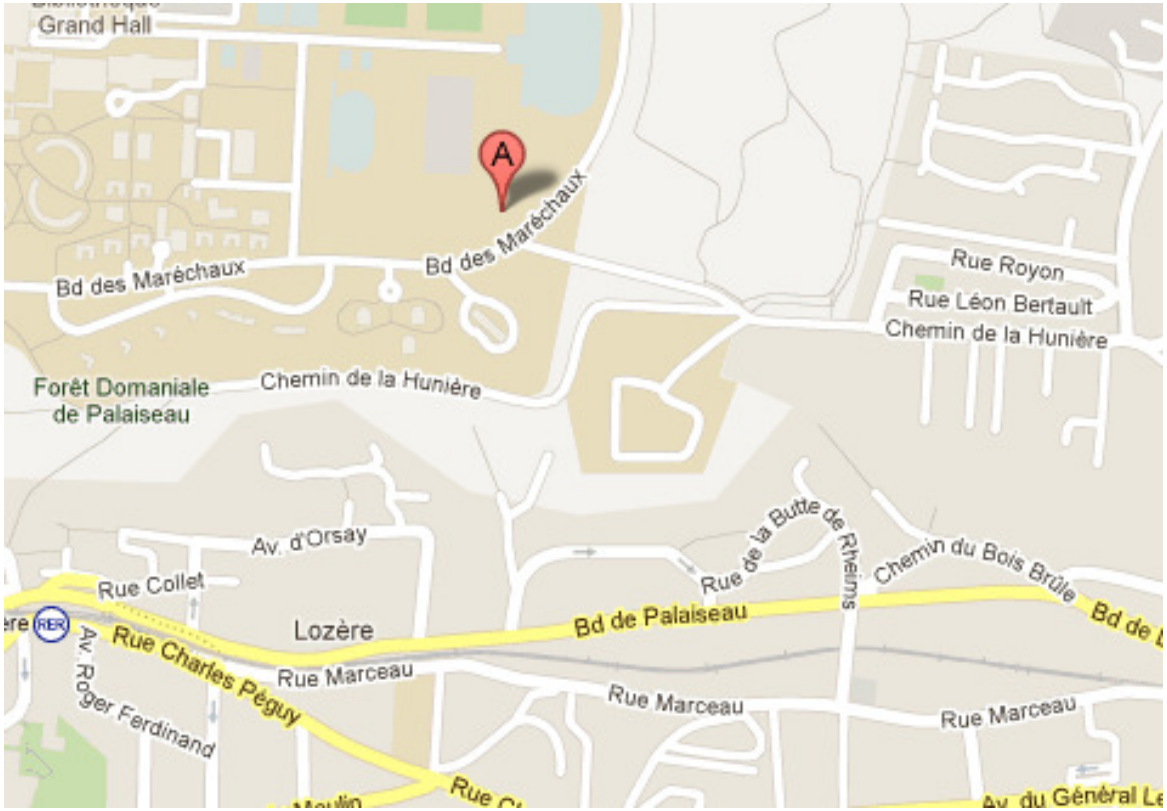
Application : génération de labyrinthe (voir TP)

Propriétés du labyrinthe

- Toutes les cases sont accessibles à partir des autres
  Structure arborescente des chemins, enracinée au niveau de l'entrée.
- Pas de cycles, c'est-à-dire, pas de création d'ilôts de murs n'étant pas reliés aux bords



Motivation – assistance au déplacement



Motivation – assistance au déplacement

Représentation discrète de l'espace

- Intersections : sommets
- Routes : arcs entre les sommets avec potentiellement des propriétés
  - une orientation (sens unique, double sens, ...)
  - une valuation ou poids (vitesse max, distance, charge, ...)
- Carte (topologique) : graphe orienté pondéré
- But : calculer un chemin avec un coût minimal par analyse du graphe

Applications des graphes

- Réseaux de communication (routes, trains, métros, télécoms . . .).
- Planification de tâches (dépendance / antériorité).
- Compilation (flot de contrôle de programme).
- Physique (chaînes de Markov).
- Robotique (carte topologique)
- Automatique (automates de contrôle).
- Biologie (réassemblage de sections de génome).
- . . .

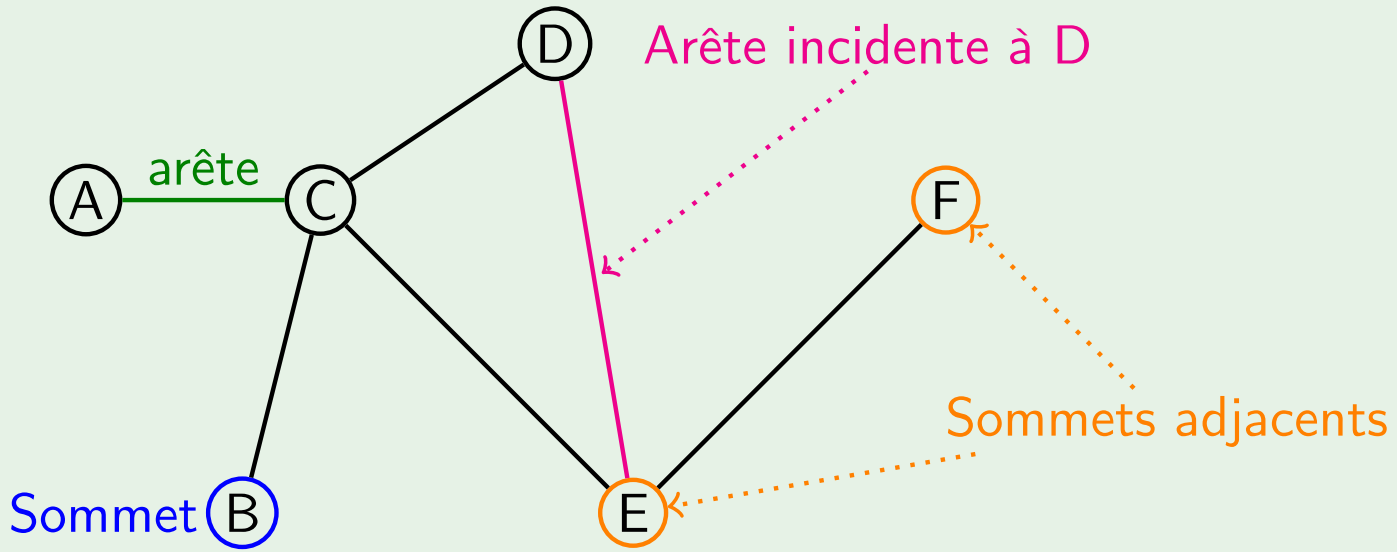
Graphes non orientés

Graphes non orientés

Un graphe non orienté (ou simplement graphe) est une paire (S, A)

- S ensemble fini de sommets (*vertex/vertices*)
- A ensemble fini d'arêtes (*edges*) sous ensemble de $S \times S$
- L'ordre d'un graphe est le nombre de sommets qui le compose.
- Les sommets a et b sont adjacents s'ils sont reliés par une arête.
- Un sommet s est incident à une arête a si s est une des deux extrémités de a, et une arête est incidente à un sommet si celui-ci est une des extrémités.

Exemple

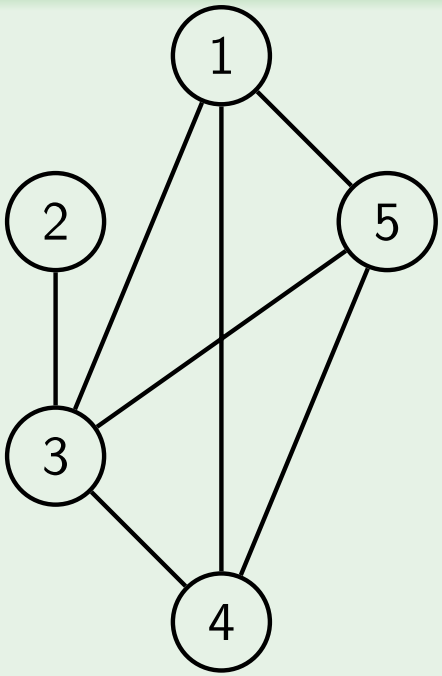


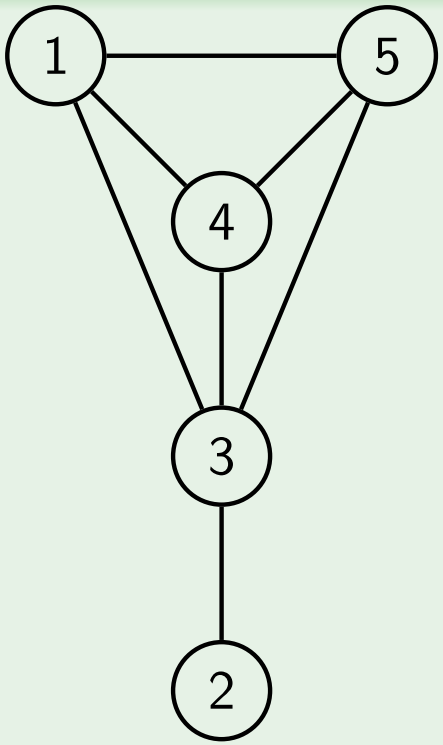
Graphes planaires

Définition

Un graphe est planaire s'il peut être dessiné dans un plan de telle manière que ses arêtes ne se coupent pas en dehors de leurs extrémités.

Exemple



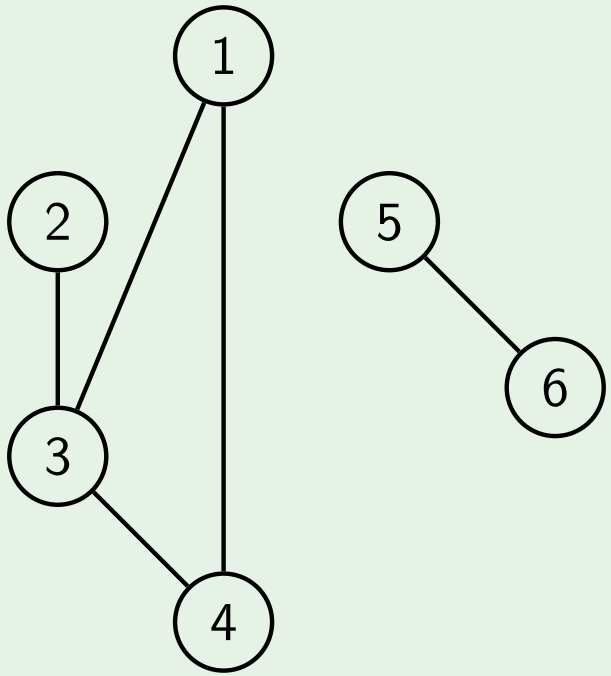


Graphes connexes

Définition

Un graphe est connexe si à partir de n'importe quel sommet il est possible de rejoindre les autres sommets en suivant les arêtes.

Exemple : graphe non connexe

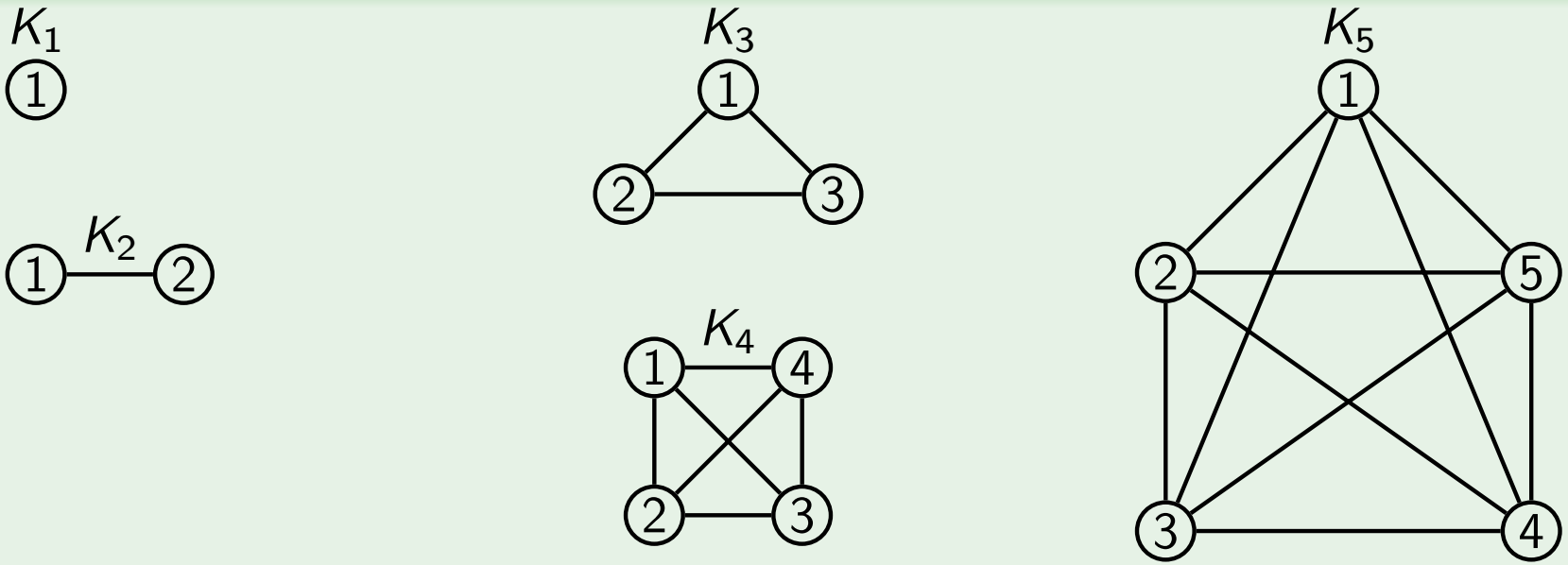


Graphes complets

Définition

Un graphe G est complet si chaque sommet de G est relié directement à tous les autres sommets.

Exemple, graphes complets de K_1 à K_5



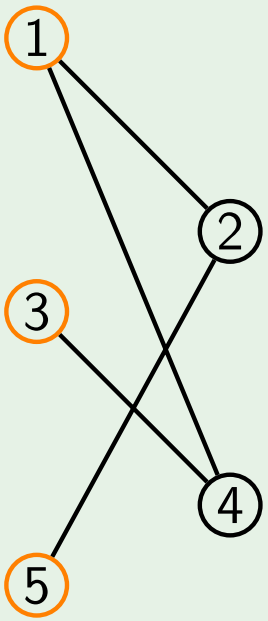
Remarque : K_5 est le plus petit graphe complet non planaire.

Graphes bipartis

Définition

Un graphe G est biparti si ses sommets S peuvent être divisés en deux sous-ensembles S_1 et S_2 telles que toutes les arêtes de G relient un sommet de S_1 à un sommet de S_2.

Exemple



- $S_1 = \{1, 3, 5\}$
- $S_2 = \{2, 4\}$

Graphes partiels et sous-graphes

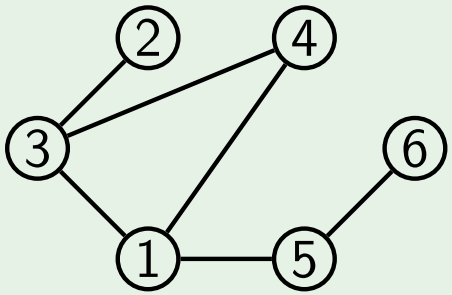
Définitions

Soit $G = (S, A)$ un graphe

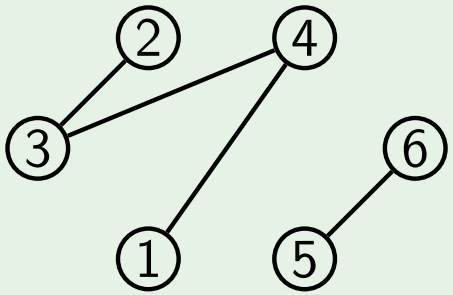
- $G' = (S, A')$ est graphe partiel de G si $A' \subseteq A$ (G' est obtenu en ôtant des arêtes de G)
- $G' = (S', A(S'))$ est un sous-graphe de G induit par $S' \subseteq S$ avec $A(S')$ l'ensemble d'arêtes de G ayant leurs extrémités dans S'.

Exemple

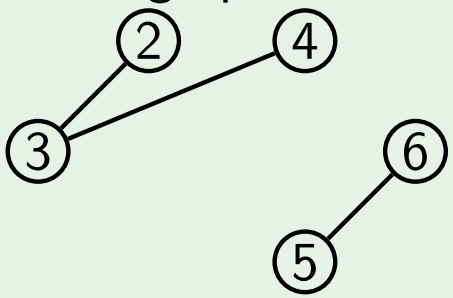
Graphe G



Graphe partiel de G



Sous graphe de G



Une clique

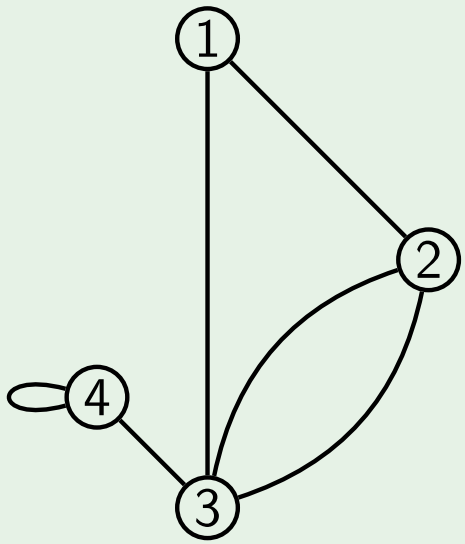
Un sous-graphe complet d'un graphe.

Graphes simples et multigraphes

Définition

Un graphe G est simple si au plus une arête lie deux sommets et il n'y a pas de boucle sur un sommet. Autrement G est un multigraphe.

Exemple de multigraphe



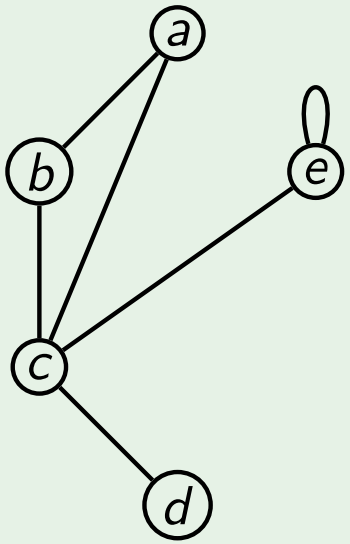
Hypothèse de travail nous nous concentrerons que sur les graphes simples dans cet enseignement.

Degrés

Définition degré d'un sommet

Le degré d'un sommet s, noté $d(s)$, est le nombre d'arêtes incidentes à ce sommet (une boucle compte pour 2).

Exemple



- $d(a) = 2$
- $d(b) = 2$
- $d(c) = 4$
- $d(d) = 1$
- $d(e) = 3$

Degré d'un graphe

Degré maximum de tous les sommets du graphe. Si tous les sommets ont un même degré k alors le graphe est k-régulier.

Chaînes

Chaîne

Une suite alternée de sommets et d'arcs, débutant et finissant par un sommet. La longueur d'une chaîne est le nombre d'arêtes la constituant.

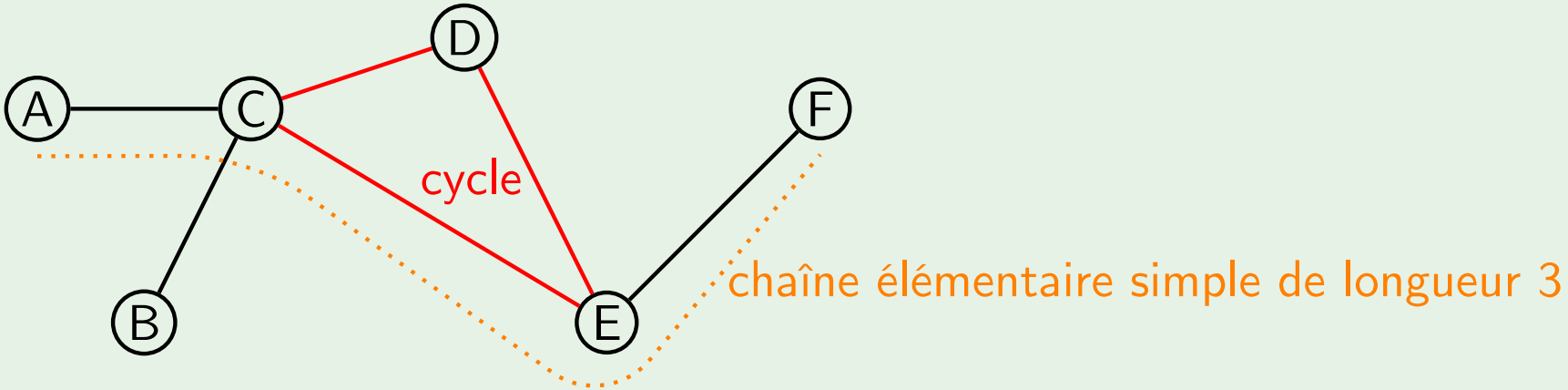
Une chaîne est

- élémentaire si chaque sommet n'y apparaît au plus une fois.
- simple si chaque arête n'y apparaît au plus une fois.
- fermée si le sommet de départ et d'arrivée est le même.

Un cycle est une chaîne simple fermée.

La distance entre deux sommets est la longueur de la plus petite chaîne les reliant.

Exemple



Graphes eulériens

Hypothèse : on considère des graphes simples connexes.

Définition : chaîne eulérienne

Une chaîne qui passe une et une seule fois par chacunes des arêtes du graphe.

Définition : cycle eulérien

Un cycle qui passe une et une seule fois par chaque arête du graphe.

Graphe eulérien

Un graphe qui possède un cycle eulérien.

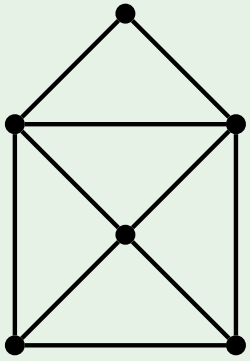
Graphe semi-eulérien

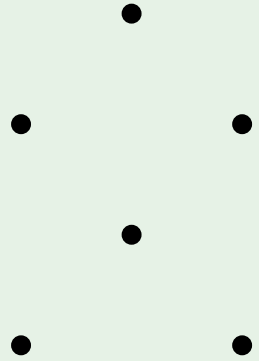
Un graphe qui possède que des chaînes eulériennes.

Graphe eulérien

Intuition : un graphe est eulérien s'il est possible de le dessiner sans lever la main.

Exemple





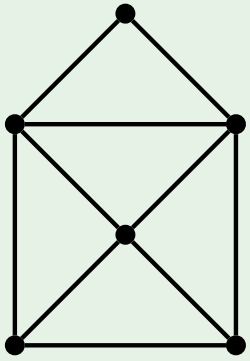
Théorèmes d'Euler

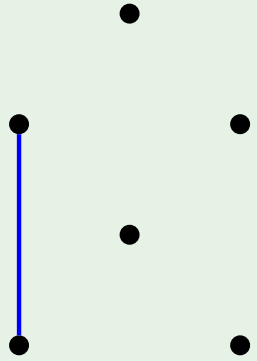
- Un graphe connexe admet une chaîne eulérienne ssi ses sommets sont tous de degré pair sauf au plus deux.
- Un graphe connexe admet un circuit eulérien ssi tous ses sommets ont un degré pair.

Graphe eulérien

Intuition : un graphe est eulérien s'il est possible de le dessiner sans lever la main.

Exemple





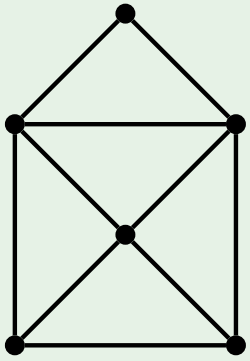
Théorèmes d'Euler

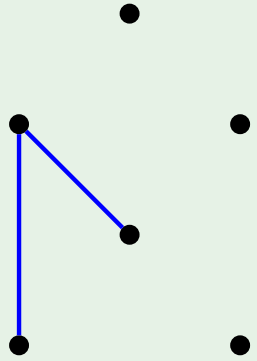
- Un graphe connexe admet une chaîne eulérienne ssi ses sommets sont tous de degré pair sauf au plus deux.
- Un graphe connexe admet un circuit eulérien ssi tous ses sommets ont un degré pair.

Graphe eulérien

Intuition : un graphe est eulérien s'il est possible de le dessiner sans lever la main.

Exemple





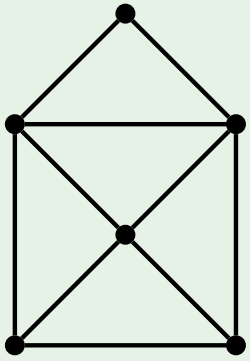
Théorèmes d'Euler

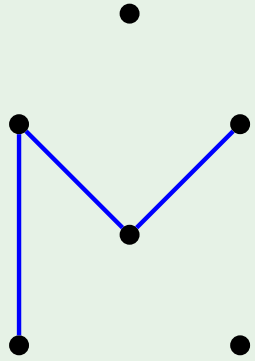
- Un graphe connexe admet une chaîne eulérienne ssi ses sommets sont tous de degré pair sauf au plus deux.
- Un graphe connexe admet un circuit eulérien ssi tous ses sommets ont un degré pair.

Graphe eulérien

Intuition : un graphe est eulérien s'il est possible de le dessiner sans lever la main.

Exemple





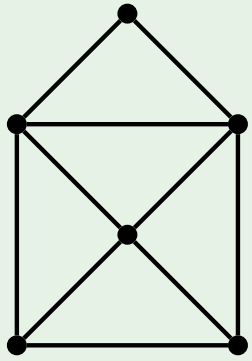
Théorèmes d'Euler

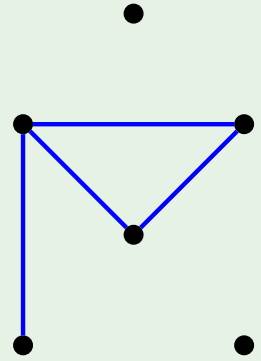
- Un graphe connexe admet une chaîne eulérienne ssi ses sommets sont tous de degré pair sauf au plus deux.
- Un graphe connexe admet un circuit eulérien ssi tous ses sommets ont un degré pair.

Graphe eulérien

Intuition : un graphe est eulérien s'il est possible de le dessiner sans lever la main.

Exemple





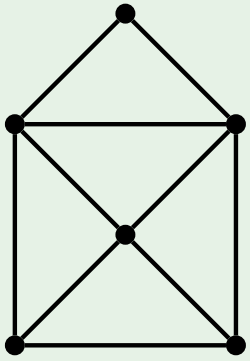
Théorèmes d'Euler

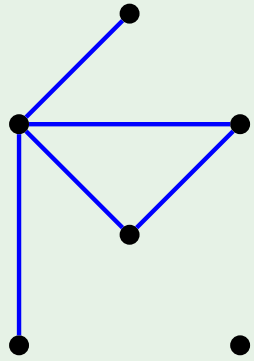
- Un graphe connexe admet une chaîne eulérienne ssi ses sommets sont tous de degré pair sauf au plus deux.
- Un graphe connexe admet un circuit eulérien ssi tous ses sommets ont un degré pair.

Graphe eulérien

Intuition : un graphe est eulérien s'il est possible de le dessiner sans lever la main.

Exemple





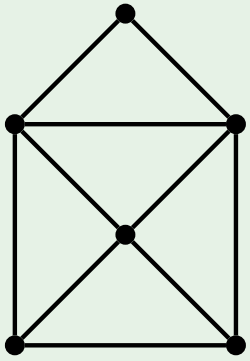
Théorèmes d'Euler

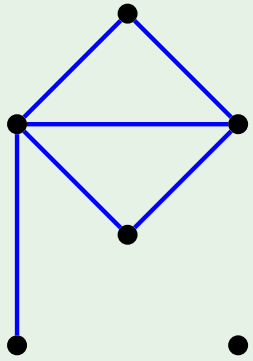
- Un graphe connexe admet une chaîne eulérienne ssi ses sommets sont tous de degré pair sauf au plus deux.
- Un graphe connexe admet un circuit eulérien ssi tous ses sommets ont un degré pair.

Graphe eulérien

Intuition : un graphe est eulérien s'il est possible de le dessiner sans lever la main.

Exemple





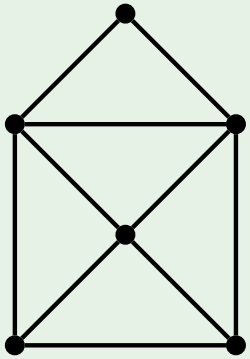
Théorèmes d'Euler

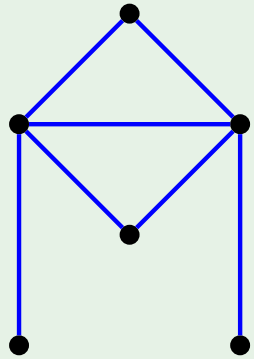
- Un graphe connexe admet une chaîne eulérienne ssi ses sommets sont tous de degré pair sauf au plus deux.
- Un graphe connexe admet un circuit eulérien ssi tous ses sommets ont un degré pair.

Graphe eulérien

Intuition : un graphe est eulérien s'il est possible de le dessiner sans lever la main.

Exemple





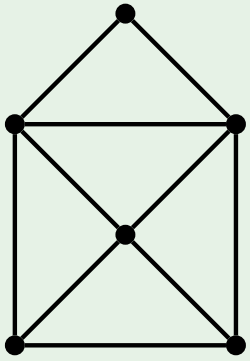
Théorèmes d'Euler

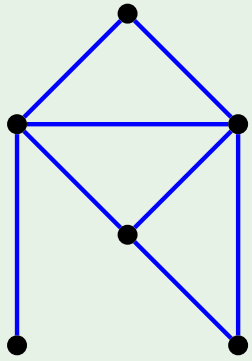
- Un graphe connexe admet une chaîne eulérienne ssi ses sommets sont tous de degré pair sauf au plus deux.
- Un graphe connexe admet un circuit eulérien ssi tous ses sommets ont un degré pair.

Graphe eulérien

Intuition : un graphe est eulérien s'il est possible de le dessiner sans lever la main.

Exemple





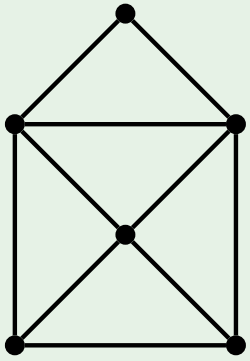
Théorèmes d'Euler

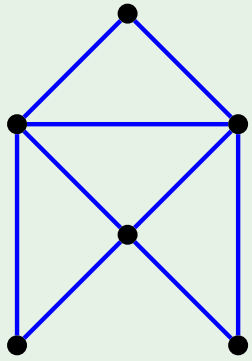
- Un graphe connexe admet une chaîne eulérienne ssi ses sommets sont tous de degré pair sauf au plus deux.
- Un graphe connexe admet un circuit eulérien ssi tous ses sommets ont un degré pair.

Graphe eulérien

Intuition : un graphe est eulérien s'il est possible de le dessiner sans lever la main.

Exemple





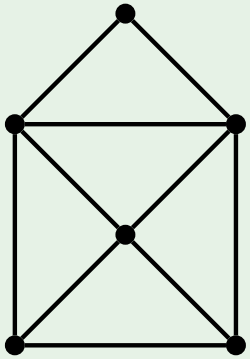
Théorèmes d'Euler

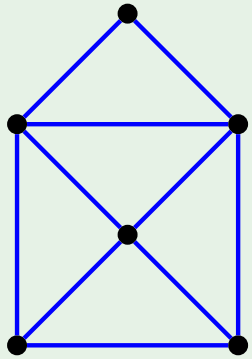
- Un graphe connexe admet une chaîne eulérienne ssi ses sommets sont tous de degré pair sauf au plus deux.
- Un graphe connexe admet un circuit eulérien ssi tous ses sommets ont un degré pair.

Graphe eulérien

Intuition : un graphe est eulérien s'il est possible de le dessiner sans lever la main.

Exemple





Théorèmes d'Euler

- Un graphe connexe admet une chaîne eulérienne ssi ses sommets sont tous de degré pair sauf au plus deux.
- Un graphe connexe admet un circuit eulérien ssi tous ses sommets ont un degré pair.

Graphes hamiltoniens

Hypothèse : on considère des graphes simples connexes.

Définition : chaîne hamiltonienne

Une chaîne qui passe une et une seule fois par chaque sommet du graphe.

Définition : cycle hamiltonien

Un cycle qui passe une et une seule fois par chacuns des sommets du graphe.

Graphe hamiltonien

Un graphe qui possède un cycle hamiltonien.

Graphe semi-hamiltonien

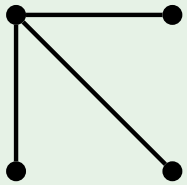
Un graphe qui possède que des chaînes hamiltoniennes.

Exemple graphe hamiltonien

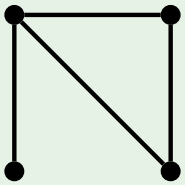
Pas de définition intuitive des graphes hamiltoniens

Exemples

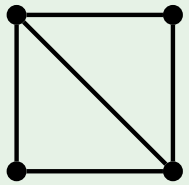
Graphe non-hamiltonien



Graphe semi-hamiltonien



Graphe hamiltonien



Pas de théorème général mais

- Théorème de Dirac un graphe simple $G = (S, A)$ à $n = |S|$ sommets ($n \geqslant 3$) avec $\forall s \in S$, $d(s) \geqslant \frac{n}{2}$ est hamiltonien.
- Théorème de Ore Un graphe simple à n sommets ($n \geqslant 3$) tel que la somme des degrés de toute paire de sommets non adjacents vaut au moins n est hamiltonien.
- ...

Nouvelles définitions pour les arbres

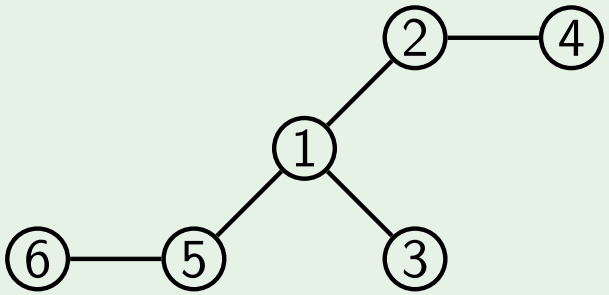
Définition

Un arbre est un graphe connexe sans cycle.

Définition

Un arbre est un graphe dans lequel deux sommets quelconques sont reliés par un et un seul chemin.

Exemple



Remarque Le nombre d'arêtes dans un arbre est égal au nombre de ses sommets moins 1

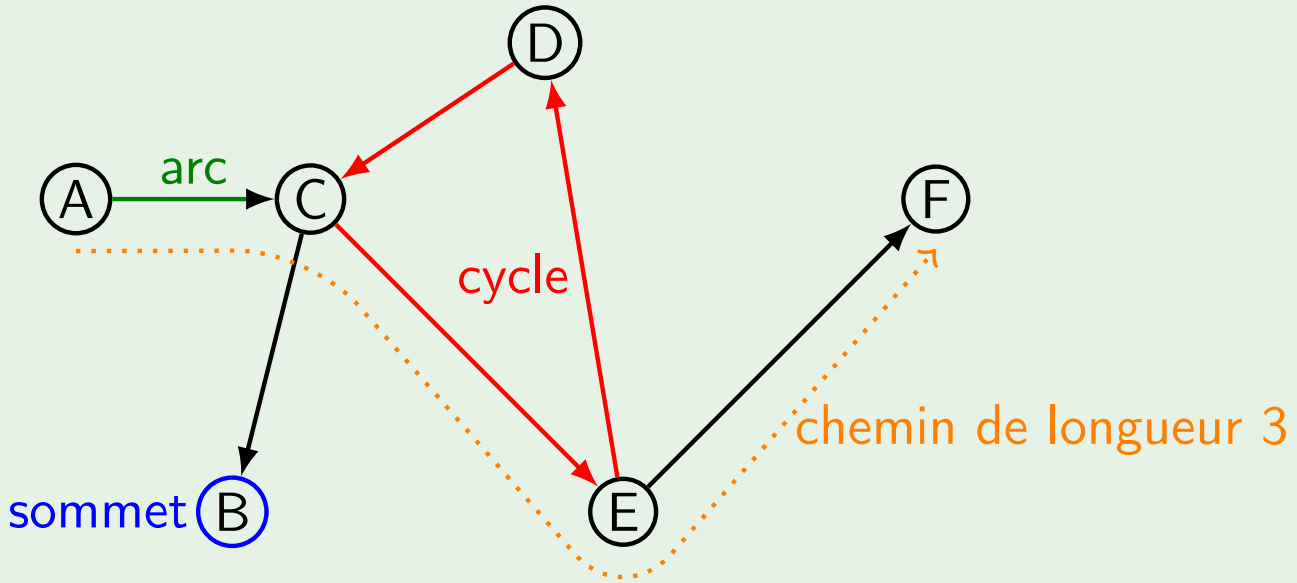
Graphes orientés

Graphes orientés

Un graphe orienté ou digraphe (*directed graph*) est une paire (S, A)

- S ensemble fini de sommets (*vertex/vertices*)
- A ensemble fini d'arcs (*arcs*) sous ensemble de $S \times S$
- un chemin (*path*) est une suite alternant sommets et arcs qui débute et termine par un sommet. Note le chemin est dirigé.
- le reste du vocabulaire de ne change pas : longueur, cycle, sommets adjacents, arc incident, eulérien, hamiltonien, ...

Exemple



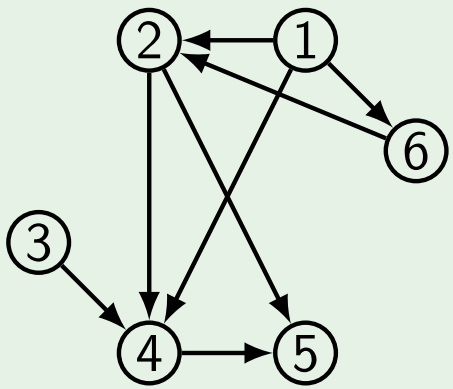
Degrés

Degré entrant et sortant

Soit s le somme d'un graphe orienté G

- le degré sortant de s, noté $d^+(s)$, est le nombre d'arcs ayant s comme origine.
- le degré entrant de s, noté $d^-(s)$, est le nombre d'arcs ayant s comme destination.
- le degré d'un sommet s, noté $d(s)$ est défini par $d^+(s) + d^-(s)$.

Exemple



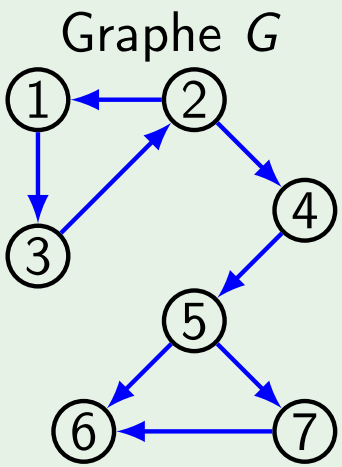
- $d^+(1) = 3$ et $d^-(1) = 0$
- $d^+(2) = 2$ et $d^-(2) = 2$
- $d^+(3) = 1$ et $d^-(3) = 0$
- $d^+(4) = 1$ et $d^-(4) = 3$
- $d^+(5) = 0$ et $d^-(5) = 2$
- $d^+(6) = 1$ et $d^-(6) = 1$

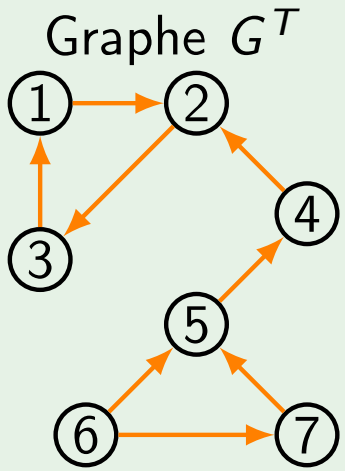
Graphe transposé

Définition

Le graphe transposé ou graphe inverse du digraphe $G = (S, A)$ est le graphe $G^T(S, A^T)$ avec $A^T = \{(y, x) : (x, y) \in A\}$.
G^T est le graphe obtenu à partir des sommets de G et dont les arcs sont dans le sens opposé à ceux de G.

Exemple





Propriétés

- $(G^T)^T = G$

Forte connexité

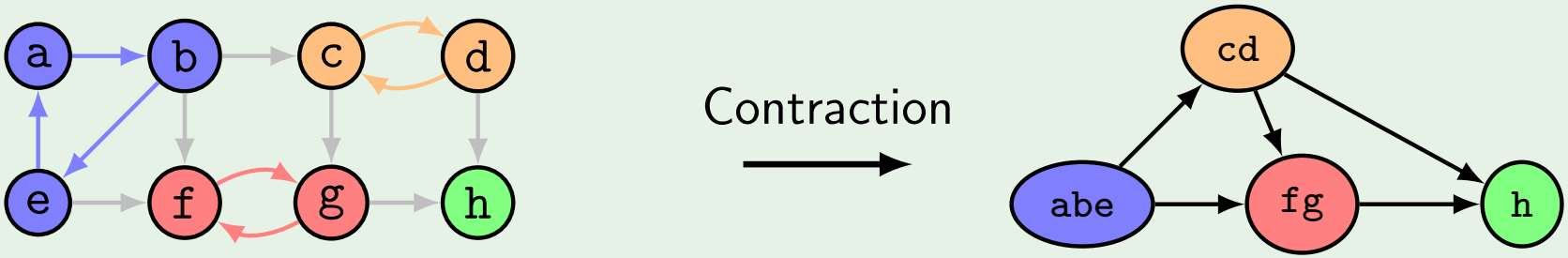
Définition : digraphe fortement connexe

Un digraphe $G = (S, A)$ est fortement connexe si pour toutes paires de sommets distincts (s_1, s_2) il existe un chemin de s_1 à s_2.

Définition : composante fortement connexe (CFC)

Une composante fortement connexe est un sous-graphe orienté G' du digraphe $G = (S, A)$ ayant la propriété de forte connexité.

Exemple



Remarque

La contraction des CFC produit nécessairement un DAG (*directed acyclic graph*)

Représentations mathématico-informatiques des (di)graphes

Représentation des graphes : matrice d'adjacence

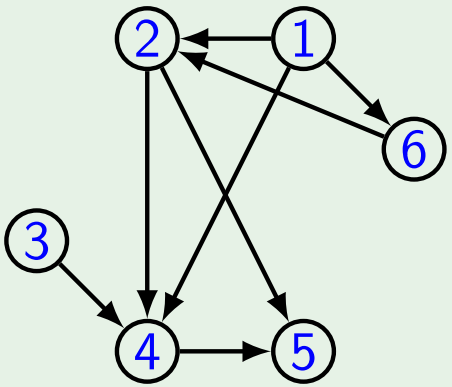
Définition

Pour un (di)graphe $G = (S, A)$, une matrice d'adjacence M est de dimension $(|S| \times |S|)$ et dont les cœfficients sont

$$\forall (i,j) \in |S| \times |S|, \quad M_{ij} = \begin{cases} 1 & \text{si} \quad (i,j) \in A \\ 0 & \text{autrement} \end{cases}$$

Remarque : dans le cas d'un graphe non orienté la matrice d'adjacence est symétrique. Dans les deux cas, les éléments diagonaux représentent les boucles.

Exemple : matrice d'adjacence d'un digraphe



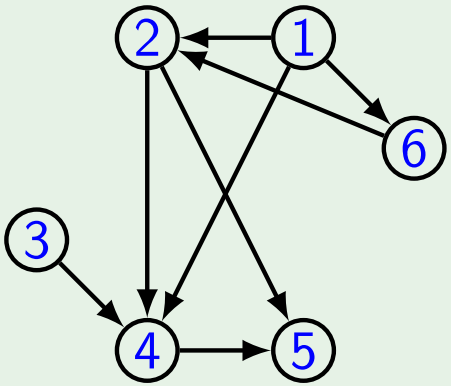
$$M = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

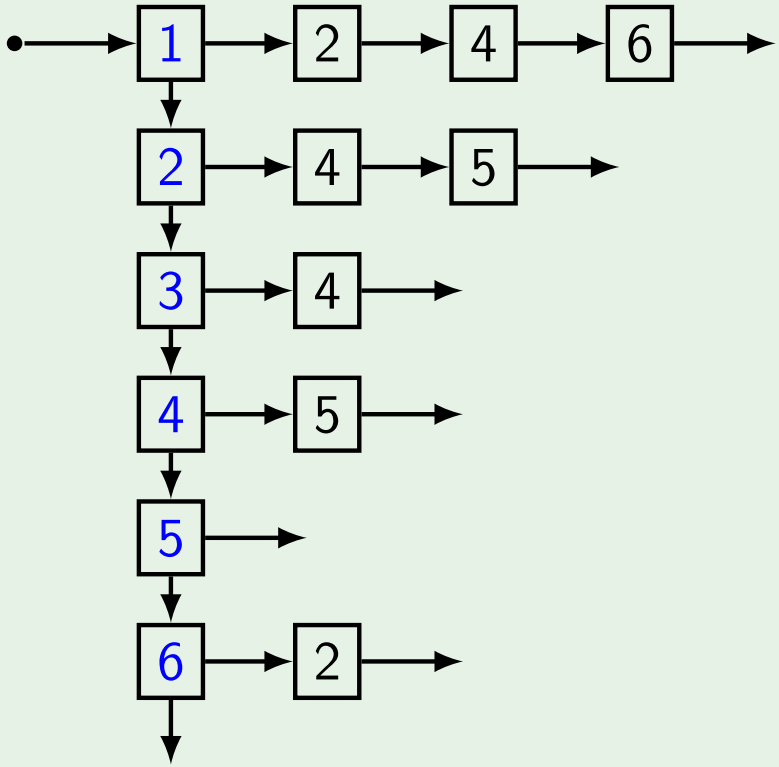
Représentation des graphes : listes d'adjacence

Définition

Structure de données fondée sur une méthode par chaînage et manipulation de pointeurs $\Rightarrow$ représentation creuse.

Exemple : liste d'adjacence d'un digraphe





Implémenation dans la bibliothèque `libin103`

Définition d'un graphe avec sommets identifiés avec des entiers

```
typedef struct integer_adjlist_elmt_ {
  int vertex;
  double weight;
} integer_adjlist_elmt_t;

typedef struct integer_adjlist_ {
  int vertex;
  generic_set_t adjacent;
} integer_adjlist_t;

typedef struct integer_graph_ {
  int vcount;
  int ecount;
  generic_list_t adjlists;
} integer_graph_t;
```

Point de vigilance !

Une liste d'adjacence est modélisée par

- une liste chaînée (générique) de structures
- dont un des champs est un ensemble (générique)
- dont les éléments sont des structures.

Remarque on considère des (di)graphes dont les arcs/arêtes peuvent avoir une valeur (`weight`). Aspect utile pour la prochaine séance.

API des graphes

API Création / Destruction

```
void integer_graph_init(integer_graph_t *g);
void integer_graph_destroy(integer_graph_t *g);
```

API Accesseurs / Prédicats

```
bool integer_graph_is_adjacent(integer_graph_t *g, int v1, int v2);
int integer_graph_vcount(integer_graph_t *g);
int integer_graph_ecount(integer_graph_t *g);
```

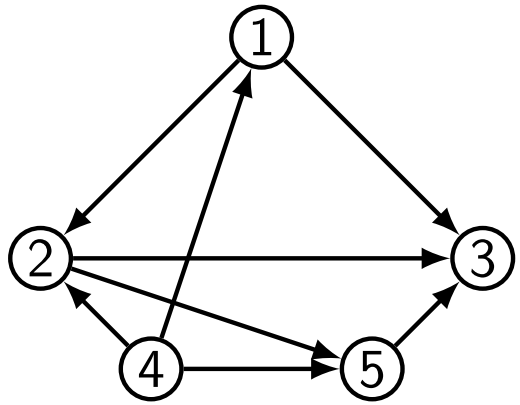
API Insertion / Suppression

```
int integer_graph_ins_vertex(integer_graph_t*, int v);
int integer_graph_ins_edge(integer_graph_t *g,
                           int v1, int v2, double w);
int integer_graph_rem_vertex(integer_graph_t *g, int v);
int integer_graph_rem_edge(integer_graph_t *g, int v1, int v2);
```

Exemple de construction d'un graphe

Code source

```
int main (int argc, char** argv) {
  integer_graph_t graph;
  integer_graph_init (&graph);

  for (int i = 1; i < 6 ; i++) {
    integer_graph_ins_vertex(&graph, i);
  }
  integer_graph_ins_edge(&graph, 1, 2, 0.0);
  integer_graph_ins_edge(&graph, 1, 3, 0.0);
  integer_graph_ins_edge(&graph, 2, 3, 0.0);
  integer_graph_ins_edge(&graph, 2, 5, 0.0);
  integer_graph_ins_edge(&graph, 4, 1, 0.0);
  integer_graph_ins_edge(&graph, 4, 5, 0.0);
  integer_graph_ins_edge(&graph, 4, 2, 0.0);
  integer_graph_ins_edge(&graph, 5, 3, 0.0);

  print_graph (&graph);
  integer_graph_destroy (&graph);
  return EXIT_SUCCESS;
}
```



Remarque

Pour représenter un graphe non orienté,

- insérer une arête et son symétrique

Complexité des opérations sur les graphes

Pour un graphe $G = (S, A)$

Fonction	Complexité	Commentaire
Initialisation	$\mathcal{O}(1)$	Mise à zéro des champs de la structure
Destruction	$\mathcal{O}(\|S\| + \|A\|)$	Pour chaque élément de la liste il faut détruire un ensemble
#Arcs / #Sommets	$\mathcal{O}(1)$	Accès direct aux champs de la structure
Test d'adjacence	$\mathcal{O}(\|S\|)$	Parcours la liste pour trouver la position
Insertion de sommet	$\mathcal{O}(\|S\|)$	Parcours de la liste pour savoir si le sommet est déjà présent
Insertion d'arc	$\mathcal{O}(\|S\|)$	On cherche dans la liste les positions des sommets
Suppression de sommet	$\mathcal{O}(\|S\| + \|A\|)$	Traverse tout le graphe pour vérifier qu'il n'y a plus d'arc impliquant le sommet
Suppression d'arc	$\mathcal{O}(\|S\|)$	On cherche dans la liste les positions du premier sommet

Fonction print_graph (le retour des itérateurs)

Code source

```
void print_graph (integer_graph_t* graph) {
  generic_list_elmt_t* elem1 =
    generic_list_head(&(graph->adjlists));
  /* Boucle sur les elements de la liste */
  for (; elem1 != NULL; elem1 = generic_list_next(elem1)) {
    integer_adjlist_t* tempV1 =
      (integer_adjlist_t*)generic_list_data(elem1);
    printf ("Vertex␣%d:␣", tempV1->vertex);

    generic_list_elmt_t* elem2 =
      generic_list_head(&(tempV1->adjacent));
    /* Boucle sur les elements de l'ensemble (set) */
    for (; elem2 != NULL; elem2 = generic_list_next(elem2)) {
      integer_adjlist_elmt_t *tempV2 =
        (integer_adjlist_elmt_t*)generic_list_data(elem2);
      printf ("%d->", tempV2->vertex);
    }
    printf ("\n");
  }
```

Parcours de graphes

Utilité des parcours de graphes

Point de départ :

- Un graphe ne sert pas à stocker des données, c'est un modèle de données représentant des relations entre données.
- On analyse le graphe pour déduire des propriétés sur les données.

Un parcours de graphe permet d'extraire des informations sur le graphe

- Prouver l'existence de ou calculer des chemin(s) avec ou sans contraintes
  - p. ex., plus court chemin entre deux sommets (cf cours 6)
- Analyse la topologie
  - Composantes (fortement pour graphe orienté) connexes (notion accessibilité).
- Étudier des notions de dépendances entre les sommets : tri topologique
- Calculer des recouvrements par des sous-graphes ou arbres (cf cours 6).
- etc.

Essentiellement deux types de parcours

- parcours en largeur
- parcours en profondeur

Parcours en largeur

Équivalent du parcours par niveaux des arbres :

- On part d'un sommet,
- on visite tous les voisins,
- on visite tous les voisins des voisins. . .

$\Rightarrow$ les sommets de distance d découverts avant ceux de distance $d+1$

Mise en œuvre

Utilisation de la technique de marquage (attribution d'une couleur) des sommets lors du parcours

- « Blanc » : non visité (ou « non marqué »).
- (« Gris » : en cours de visite, si besoin.)
- « Noir » : déjà visité (ou « marqué »).

pour éviter de boucler (c.-à-.d, éviter de visiter plusieurs fois le même sommet)

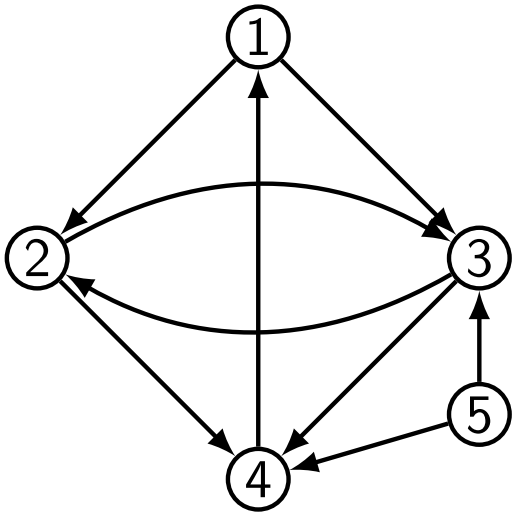
Parcours en largeur

Algorithme

```
Function bfs(graph, racine) :
    queue_t q;
    queue_enqueue(q, racine);
    set_t s;
    set_insert(s, racine);
    while queue_size(q) > 0 do
        elem = queue_dequeue(q);
        /* Traiter elem */
        for /* Pour tous les voisins de elem */ do
            if set_is_member(s, elem) == false then
                set_insert(s, elem);
                queue_enqueue(q, elem);
            end
        end
    end
```

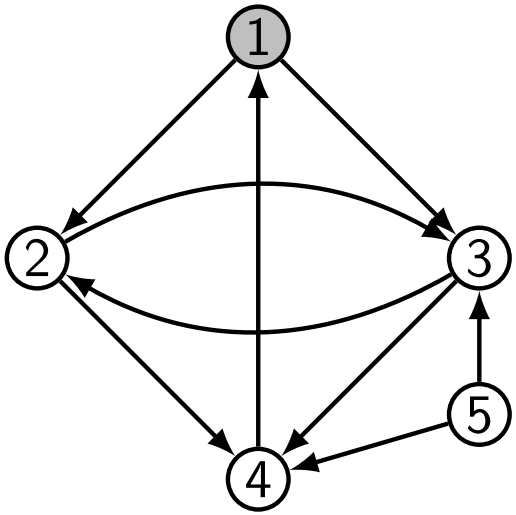
- utilisation d'une file pour stocker les voisins
- utilisation d'un ensemble pour « marquer » les sommets visités
- Complexité en $\mathcal{O}(|S| + |A|)$ avec `libin103`.

Parcours en largeur – Exemple



Initialement tous les sommets sont « blancs »

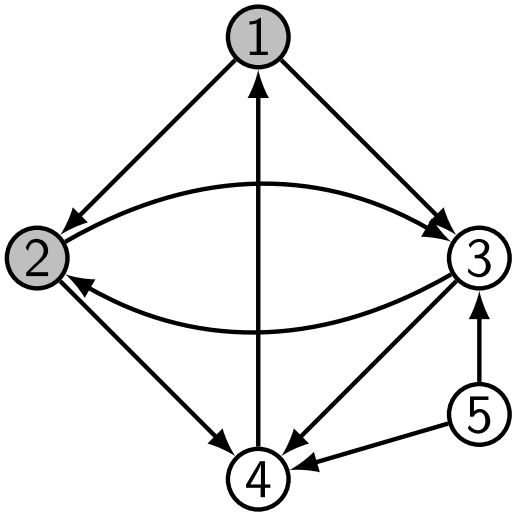
Parcours en largeur – Exemple



Initialement tous les sommets sont « blancs »

1. On commence par le sommet 1

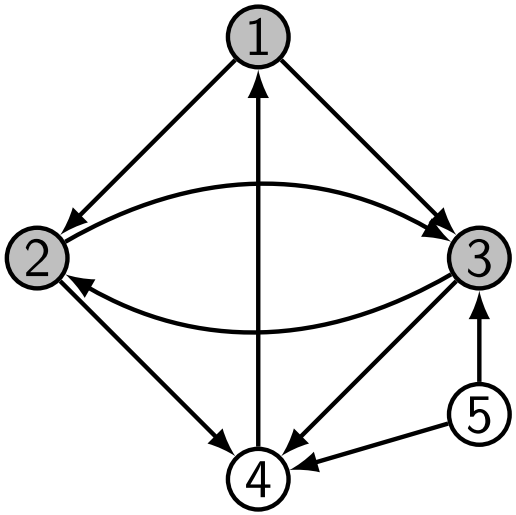
Parcours en largeur – Exemple



Initialement tous les sommets sont « blancs »

1. On commence par le sommet 1
2. sommet 2 dans la file

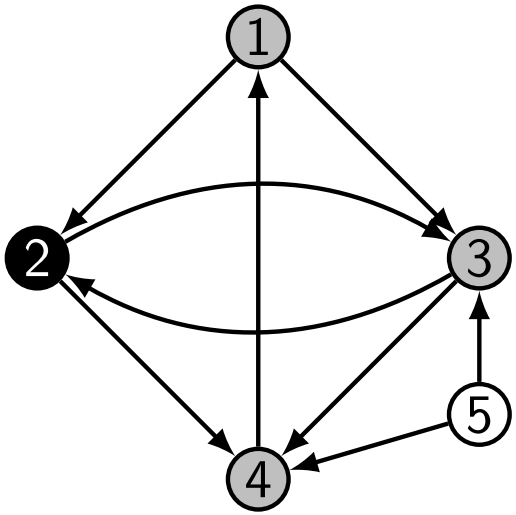
Parcours en largeur – Exemple



Initialement tous les sommets sont « blancs »

1. On commence par le sommet 1
2. sommet 2 dans la file
3. sommet 3 dans la file

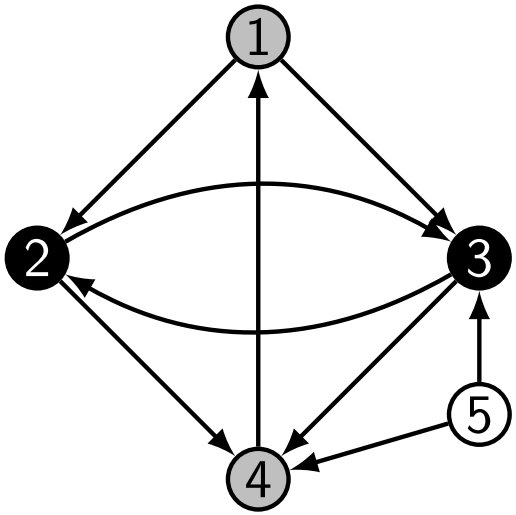
Parcours en largeur – Exemple



Initialement tous les sommets sont « blancs »

1. On commence par le sommet 1
2. sommet 2 dans la file
3. sommet 3 dans la file
4. sommet 4 dans la file et fin traitement sommet 2

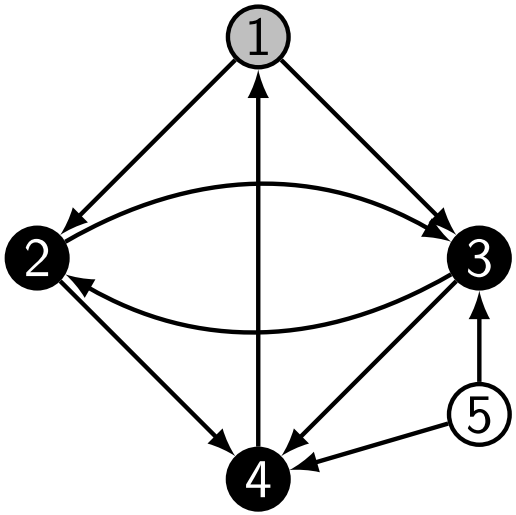
Parcours en largeur – Exemple



Initialement tous les sommets sont « blancs »

1. On commence par le sommet 1
2. sommet 2 dans la file
3. sommet 3 dans la file
4. sommet 4 dans la file et fin traitement sommet 2
5. fin traitement sommet 3

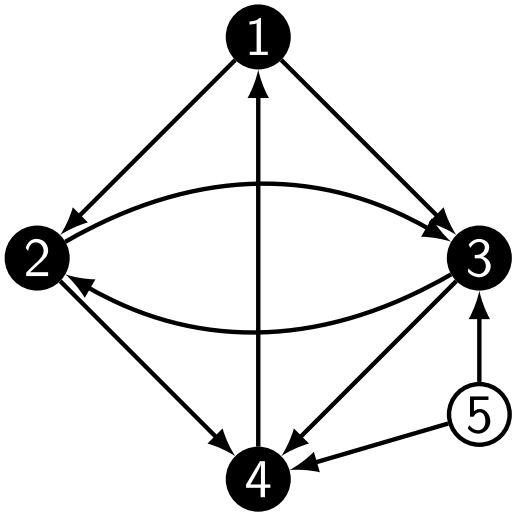
Parcours en largeur – Exemple



Initialement tous les sommets sont « blancs »

1. On commence par le sommet 1
2. sommet 2 dans la file
3. sommet 3 dans la file
4. sommet 4 dans la file et fin traitement sommet 2
5. fin traitement sommet 3
6. fin traitement sommet 4

Parcours en largeur – Exemple



Initialement tous les sommets sont « blancs »

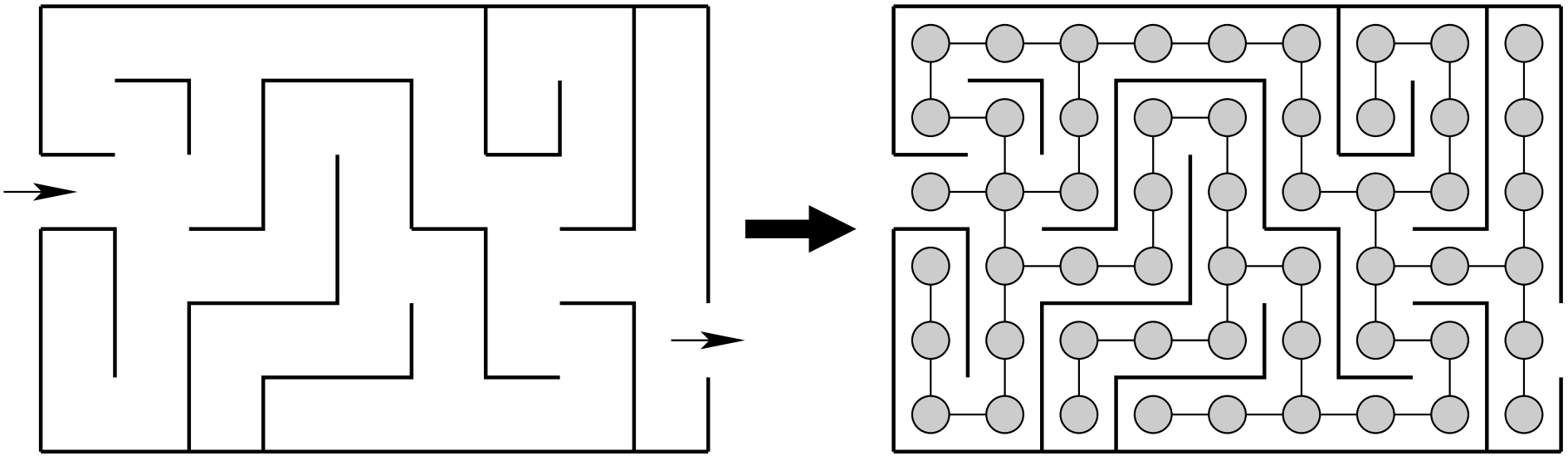
1. On commence par le sommet 1
2. sommet 2 dans la file
3. sommet 3 dans la file
4. sommet 4 dans la file et fin traitement sommet 2
5. fin traitement sommet 3
6. fin traitement sommet 4
7. fin traitement sommet 1

Remarque

- On peut avoir une couverture partielle du graphe en fonction du sommet de départ
- L'ordre de visite des sommets peut dépendre de l'ordre de construction du graphe

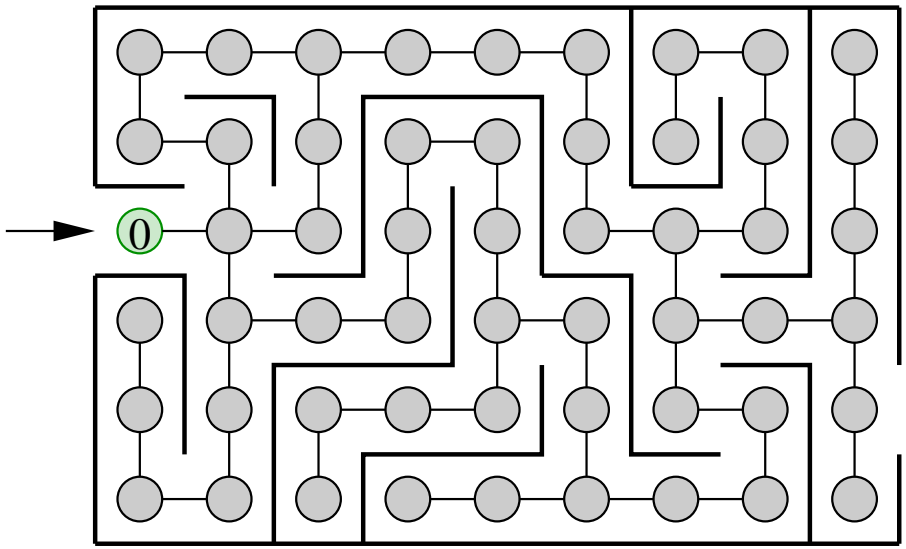
Parcours en largeur – application à un labyrinthe – 1

Objectif trouver (le plus court chemin) vers la sortie



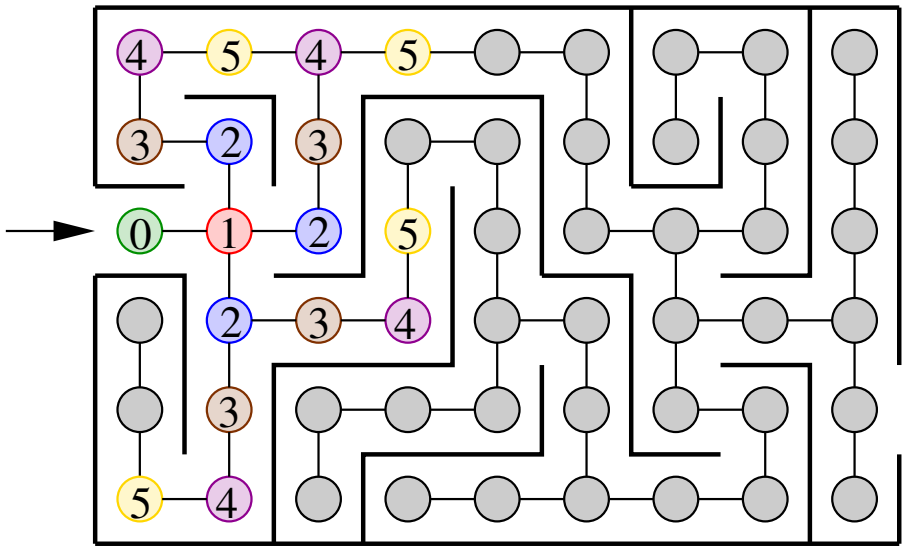
Parcours en largeur – application à un labyrinthe – 2

On commence le parcours par l'entrée (sommet vert).



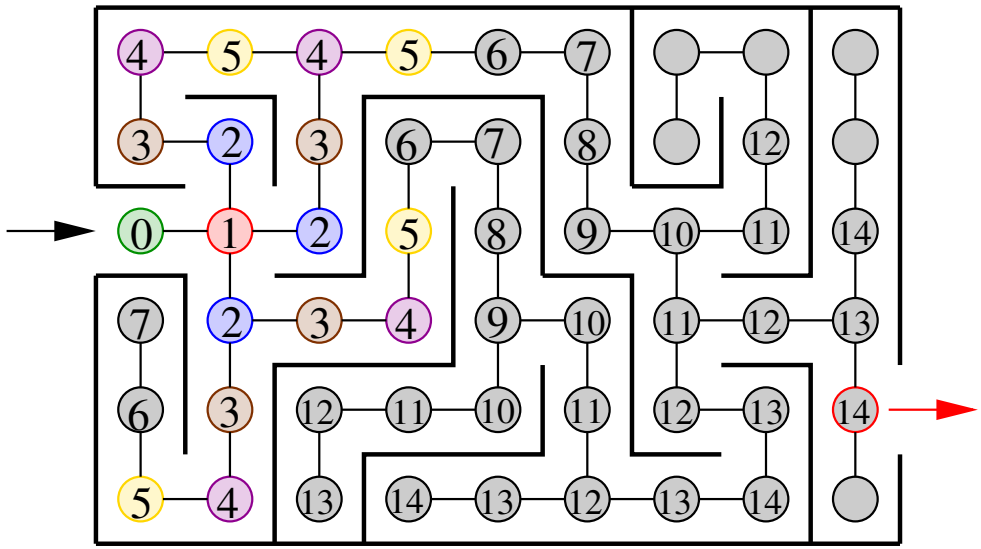
Parcours en largeur – application à un labyrinthe – 3

On descend en largeur en mémorisant le père de chaque sommet visité.



Parcours en largeur – application à un labyrinthe – 4

Arrivé à la sortie, on retrace le chemin en remontant les pères.



Parcours en profondeur

Équivalent du parcours en profondeur des arbres :

- On descend au plus profond en premier.
- Algorithme naturellement récursif.

$\Rightarrow$ aucune garantie de plus court chemin.

Mise en œuvre

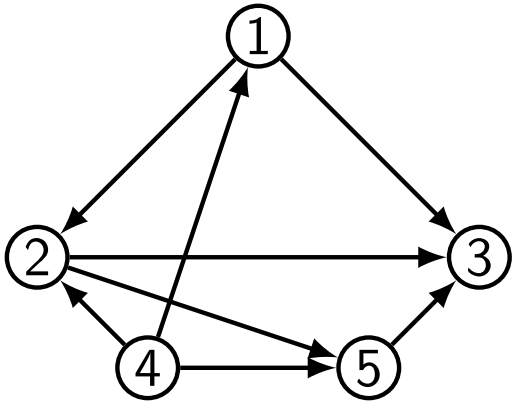
Comme pour le parcours en largeur, on veut éviter les boucles $\Rightarrow$ technique de marquage.

Algorithme

```
Function dfs(graph, sommet) :
    marquer(sommet);
    /* pre-traiter(sommet) */
    for /* Pour tous les voisins v de elem non marques */ do
        dfs(graph, v);
    end
    /* post-traiter(sommet) */
```

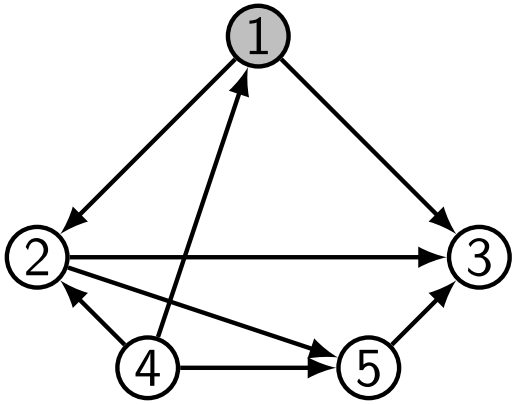
- Complexité en $\mathcal{O}(|S| + |A|)$ avec `libin103`

Parcours en profondeur – Exemple



Initialement tous les sommets sont « blancs »

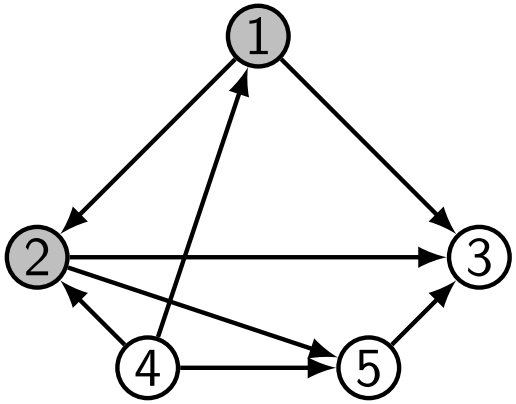
Parcours en profondeur – Exemple



Initialement tous les sommets sont « blancs »

1. On commence par le sommet 1

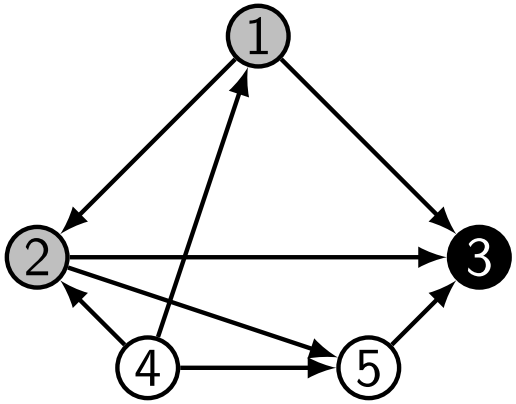
Parcours en profondeur – Exemple



Initialement tous les sommets sont « blancs »

1. On commence par le sommet 1
2. puis le sommet 2

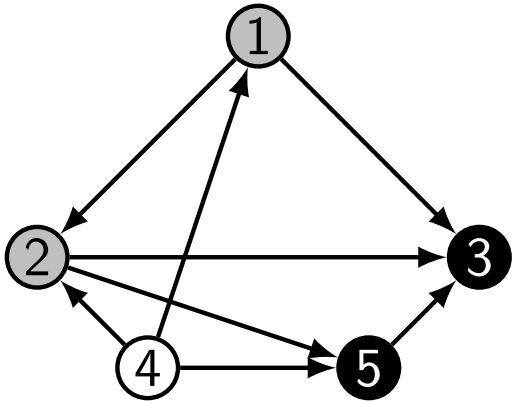
Parcours en profondeur – Exemple



Initialement tous les sommets sont « blancs »

1. On commence par le sommet 1
2. puis le sommet 2
3. puis le sommet 3 (et fin)

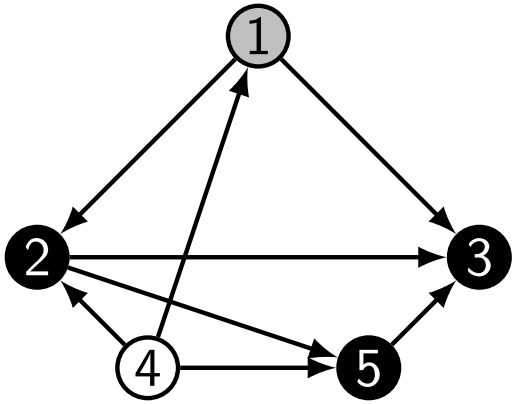
Parcours en profondeur – Exemple



Initialement tous les sommets sont « blancs »

1. On commence par le sommet 1
2. puis le sommet 2
3. puis le sommet 3 (et fin)
4. puis le sommet 5 (et fin)

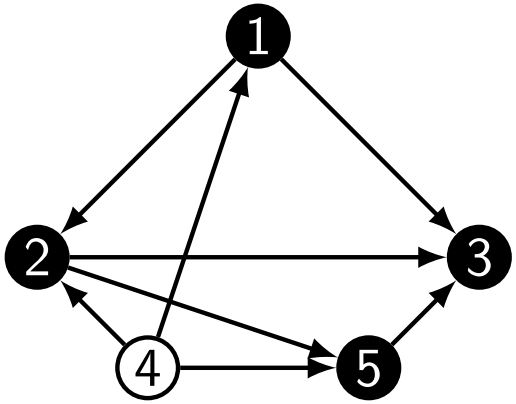
Parcours en profondeur – Exemple



Initialement tous les sommets sont « blancs »

1. On commence par le sommet 1
2. puis le sommet 2
3. puis le sommet 3 (et fin)
4. puis le sommet 5 (et fin)
5. fin traitement sommet 2

Parcours en profondeur – Exemple



Initialement tous les sommets sont « blancs »

1. On commence par le sommet 1
2. puis le sommet 2
3. puis le sommet 3 (et fin)
4. puis le sommet 5 (et fin)
5. fin traitement sommet 2
6. fin traitement sommet 1

Remarques

- On peut avoir une couverture partielle du graphe en fonction du sommet de départ
- L'ordre de visite des sommets peut dépendre de l'ordre de construction du graphe

Parcours en profondeur – extension

On peut étendre le parcours en profondeur pour garder une notion de temporalité

- enregistré les « dates » de début et de fin de visite.

Algorithme

```
Function dfs(graph, sommet) :
    marquer(sommet);
    debut[sommet] = cpt;
    cpt = cpt + 1;
    /* pre-traiter(sommet) */
    for /* Pour tous les voisins v de elem non marques */ do
        parent[v] = sommet;
        dfs(graph, v);
    end
    fin[sommet] = cpt;
    cpt = cpt + 1;
    /* post-traiter(sommet) */
```

Remarque

Dans un graphe sans cycle, les dates de fin de visite définissent une relation d'ordre sur les sommets : si il y a un chemin de u à v alors $\texttt{fin}[u] > \texttt{fin}[v]$.

Parcours en profondeur – application tri topologique

Objectif : un ensemble de tâches à ordonner selon une contrainte de précédence (certaines tâches sont à réaliser avant d'autres). Par exemple,

- l'ordre de compilation dans les `Makefile`
- gestion de projets

Modèle

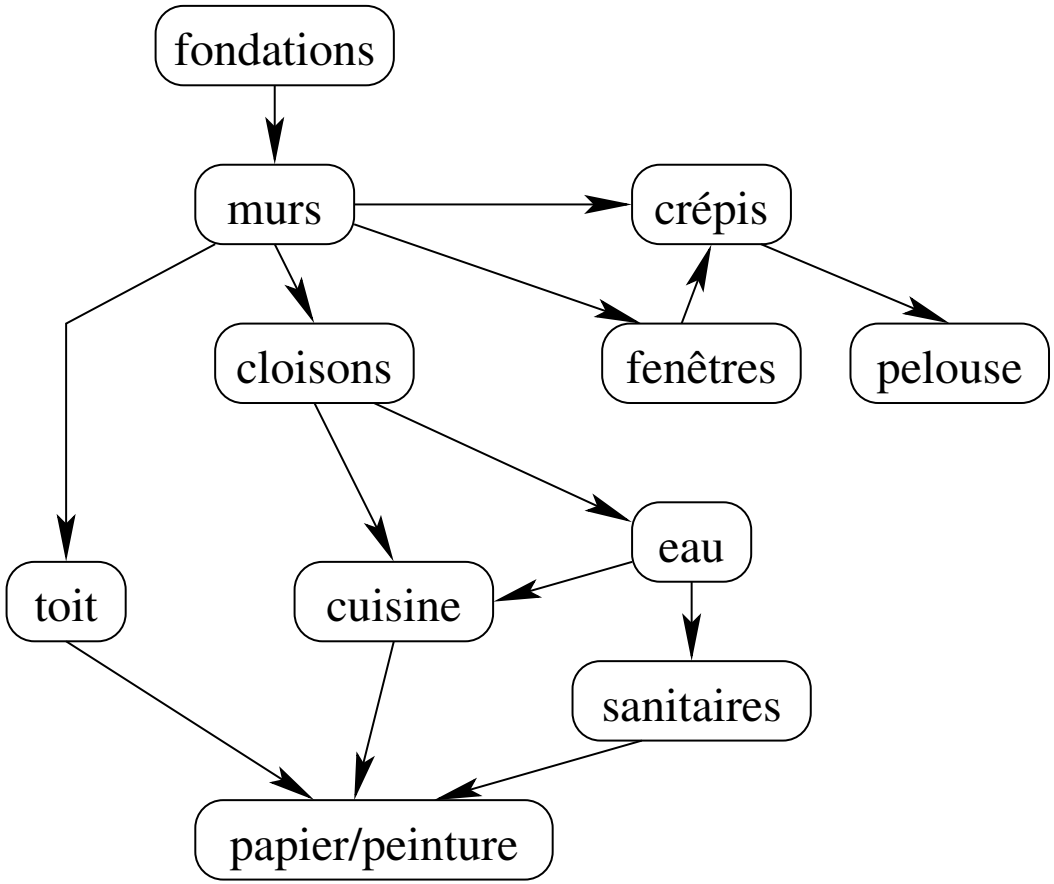
Le graphe modélise les dépendances entre les tâches.
Hypothèse : le graphe est sans cycle

Idée de l'algorithme

- Effectuer un parcours en profondeur du graphe et enregistrer date de fin
- Retourner les sommets dans l'ordre décroissant des dates de fin

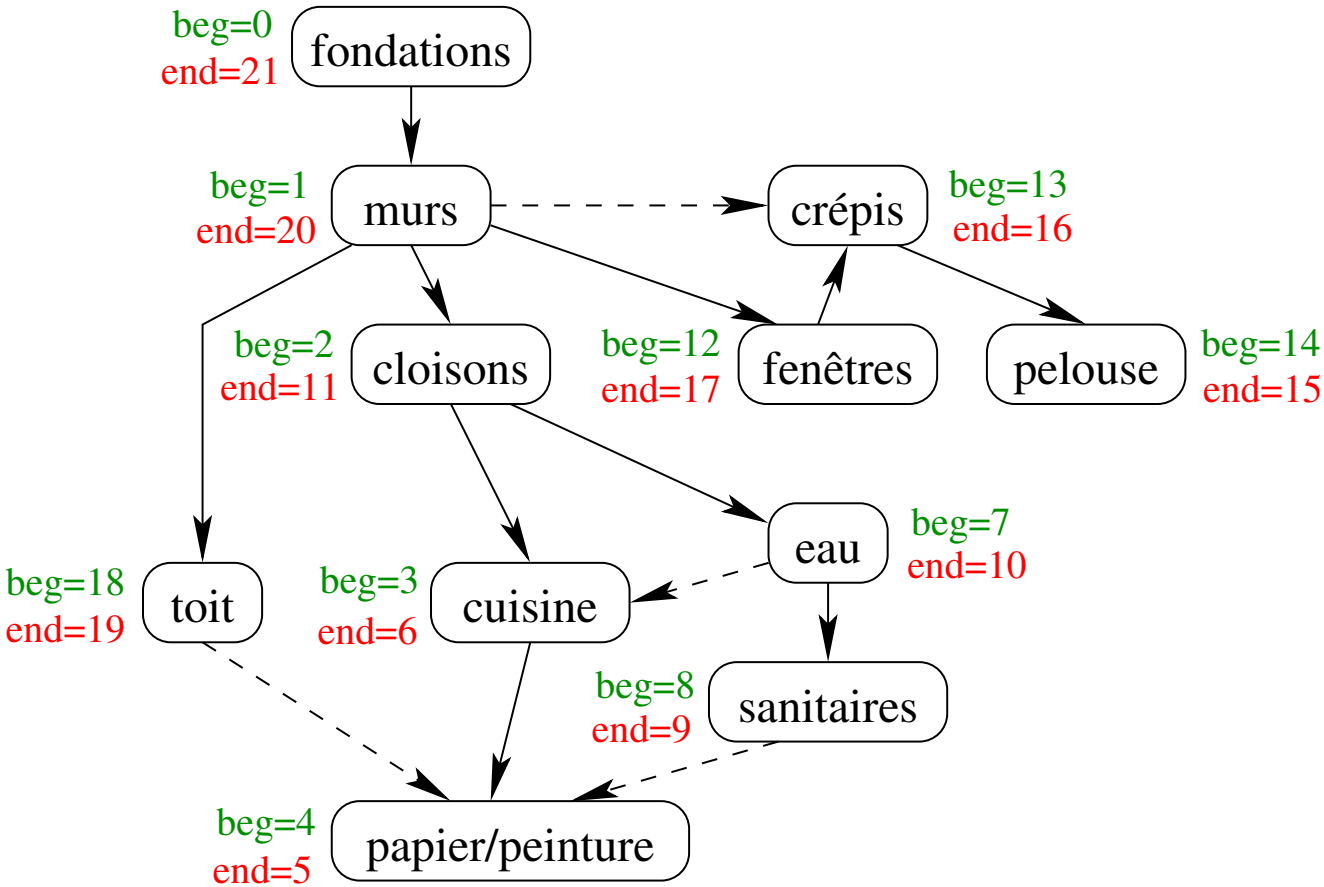
Parcours en profondeur – application tri topologique

Décomposition des tâches pour construire une maison, les arcs d'un graphe décrivent la relation faire u puis v.



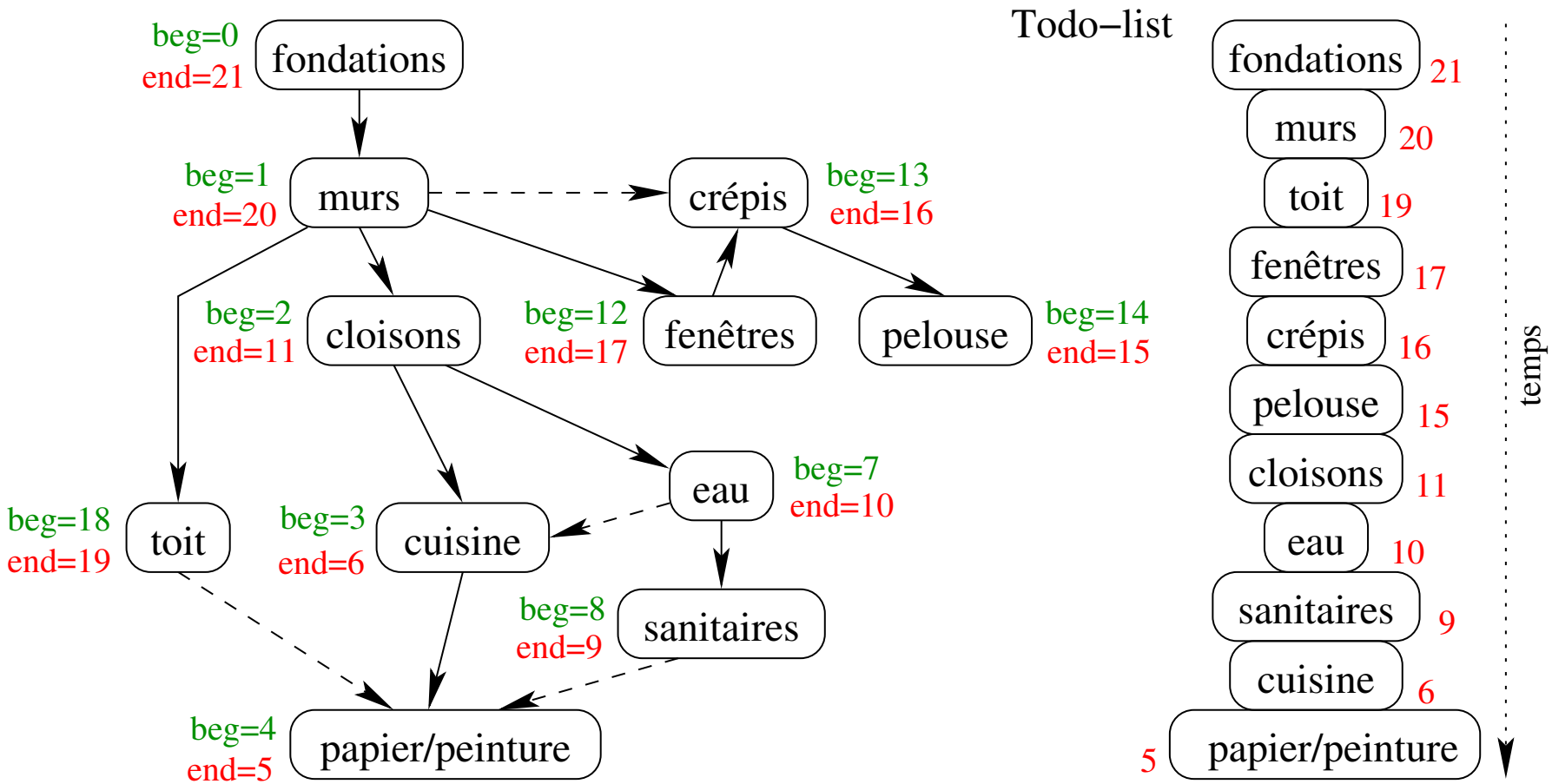
Parcours en profondeur – application tri topologique

Application du parcours en profondeur



Parcours en profondeur – application tri topologique

Tâches à effectuer en ordre décroissant des dates de fin de visite.



Connectivité des (di)graphes

Notion de connexité

Exemple d'application : réseaux sociaux

Les liens entre les différents membres peuvent être modélisés par

- un graphe dirigé (digraphe), par exemple, pour Twitter ou Strava. Une personne peut suivre le fil d'une autre sans réciprocité.
- un graphe non dirigé, par exemple, pour Facebook. Deux personnes sont dans le même groupe d'amis.

Définition

Un (di)graphe est (fortement) connecté si entre toutes paires de sommets (s_1, s_2) il existe un (chemin) chaîne.

Vocabulaire, pour un graphe $G = (S, A)$

- une composante est un sous-graphe maximal connecté de G.
- Un point d'articulation de G est un sommet dont la suppression augmente le nombre de composantes connexes.
- Un isthme est une arête dont la suppression a le même effet.

Calcul des composantes connexes d'un graphe non orienté

Idée

Soit $G = (S, A)$ un graphe (non orienté), on souhaite calculer toutes les composantes connexes d'un graphe.

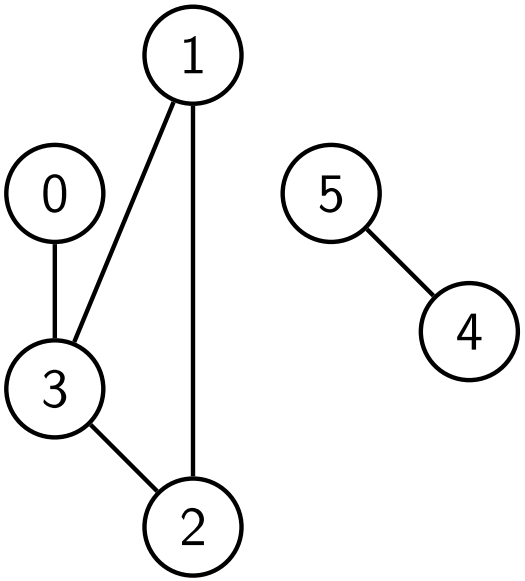
Solution facile avec l'utilisation d'une structure de données `union-find`.

Pseudo-code

```
Function cc(graph G) :
    uf_init (&dset, |S|);
    for Tous les sommets s de S do
        uf_add_element (&dset, s);
    end
    for Toutes les arêtes (u,v) de A do
        if not uf_are_connected(&dset, u, v) then
            uf_union (&dset, u, v);
        end
    end
```

Déroulement de l'algorithme

Graphe G



Structure de données `union-find`

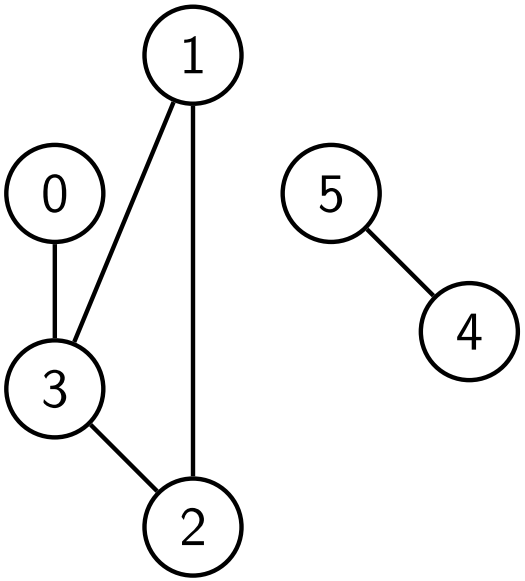
0	1	2	3	4	5
NULL	NULL	NULL	NULL	NULL	NULL

Principales étapes

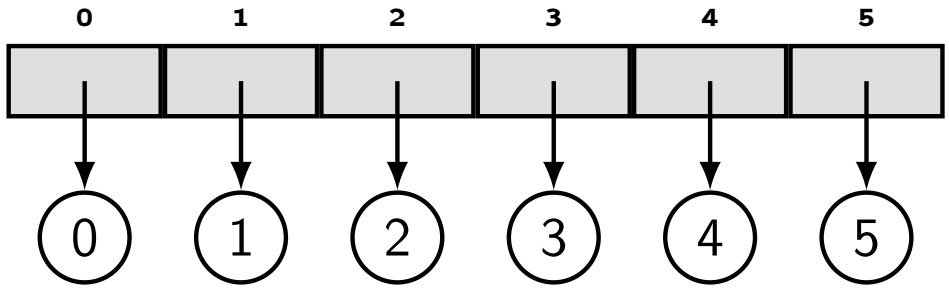
- Déclaration et initialisatione de la structure de données `union-find`

Déroulement de l'algorithme

Graphe G



Structure de données `union-find`

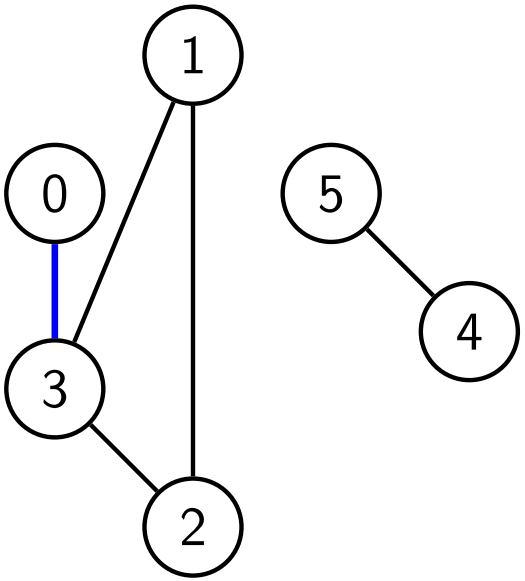


Principales étapes

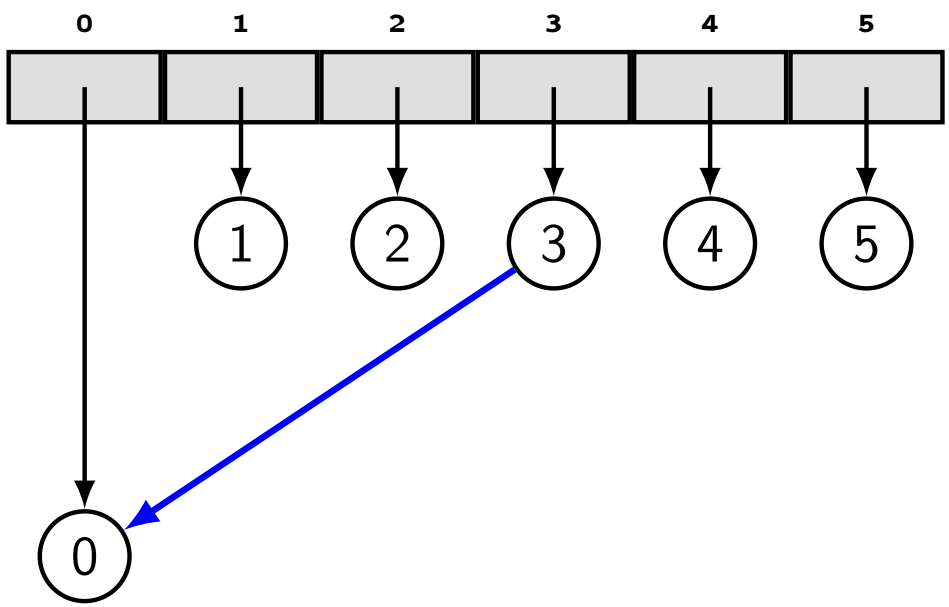
- Déclaration et initialisatione de la structure de données `union-find`
- Ajouts de tous les sommets de G dans `union-find` (première boucle `for`)

Déroulement de l'algorithme

Graphe G



Structure de données `union-find`



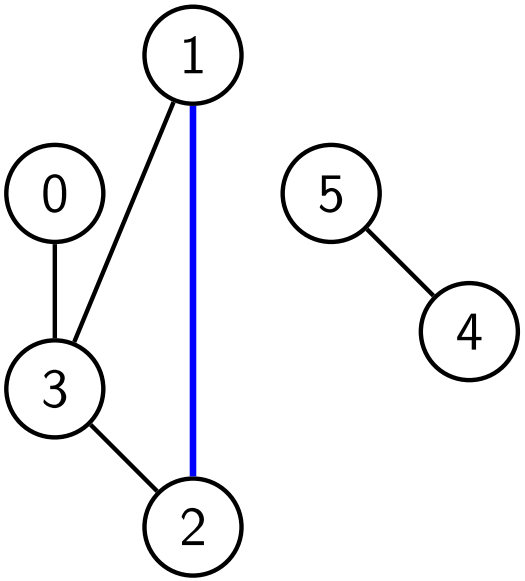
Principales étapes

- Déclaration et initialisatione de la structure de données `union-find`
- Ajouts de tous les sommets de G dans `union-find` (première boucle `for`)
- Traitement des arcs de G et potentiels unions (seconde boucle `for`)

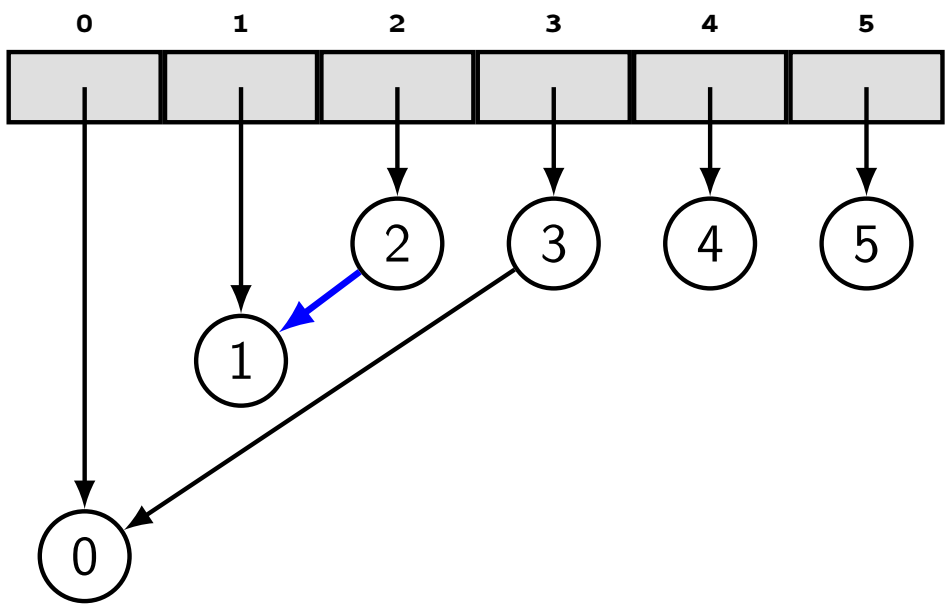
$$(0,3),\ (1,2),\ (1,3),\ (2,3),\ (5,4)$$

Déroulement de l'algorithme

Graphe G



Structure de données `union-find`



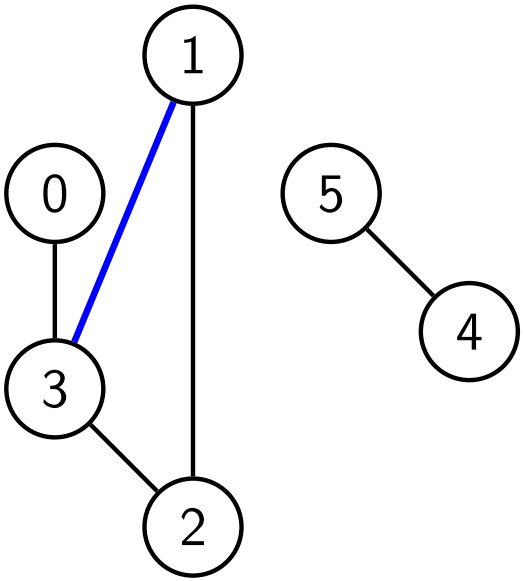
Principales étapes

- Déclaration et initialisatione de la structure de données `union-find`
- Ajouts de tous les sommets de G dans `union-find` (première boucle `for`)
- Traitement des arcs de G et potentiels unions (seconde boucle `for`)

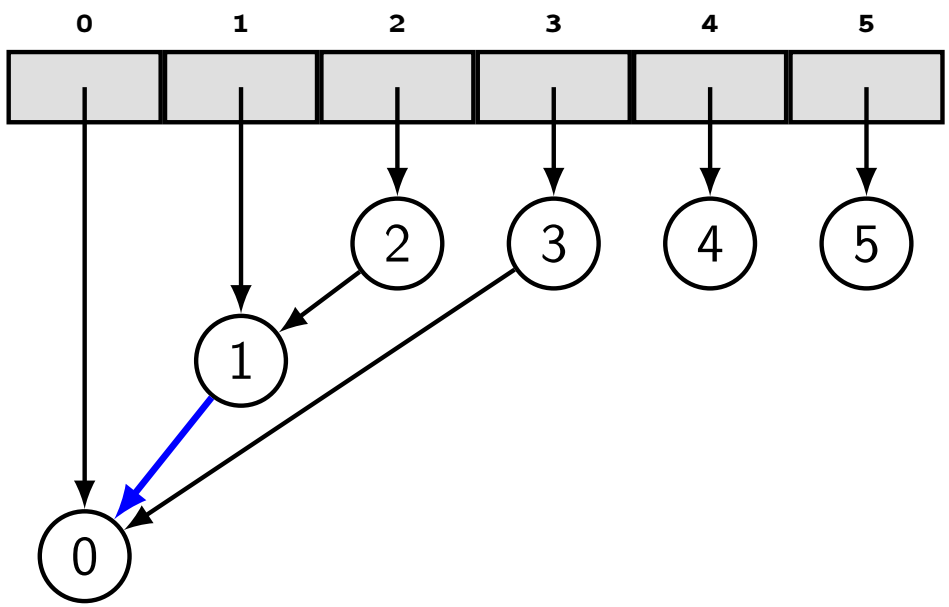
$$(0,3),\ (1,2),\ (1,3),\ (2,3),\ (5,4)$$

Déroulement de l'algorithme

Graphe G



Structure de données `union-find`



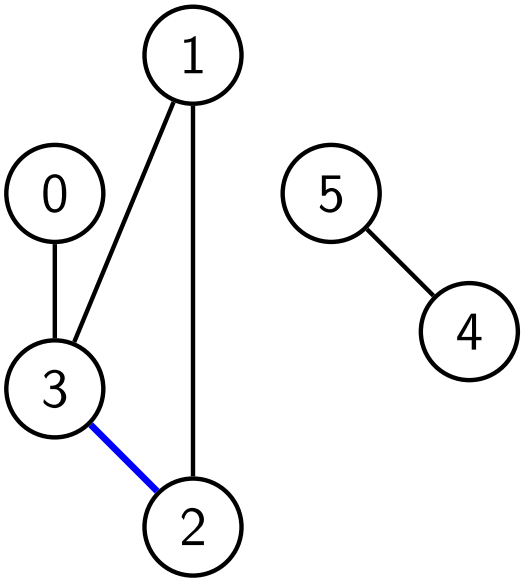
Principales étapes

- Déclaration et initialisatione de la structure de données `union-find`
- Ajouts de tous les sommets de G dans `union-find` (première boucle `for`)
- Traitement des arcs de G et potentiels unions (seconde boucle `for`)

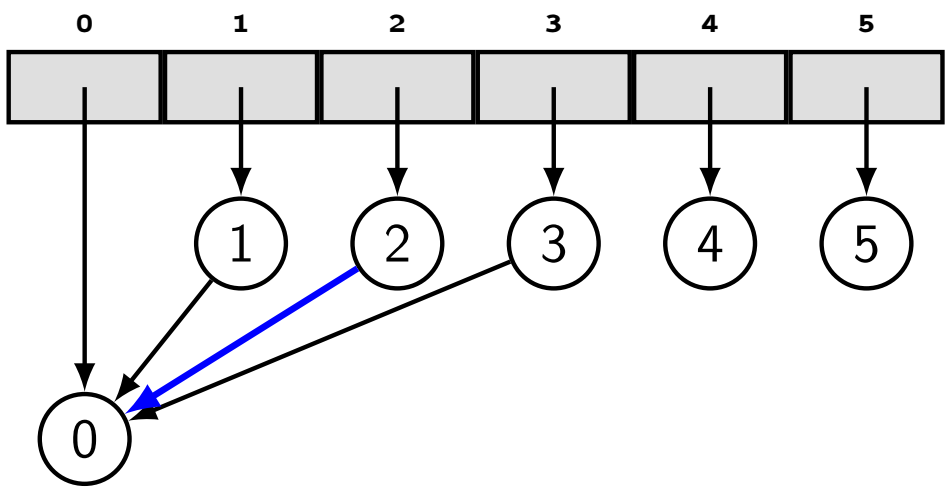
$$(0,3),\ (1,2),\ (1,3),\ (2,3),\ (5,4)$$

Déroulement de l'algorithme

Graphe G



Structure de données `union-find`



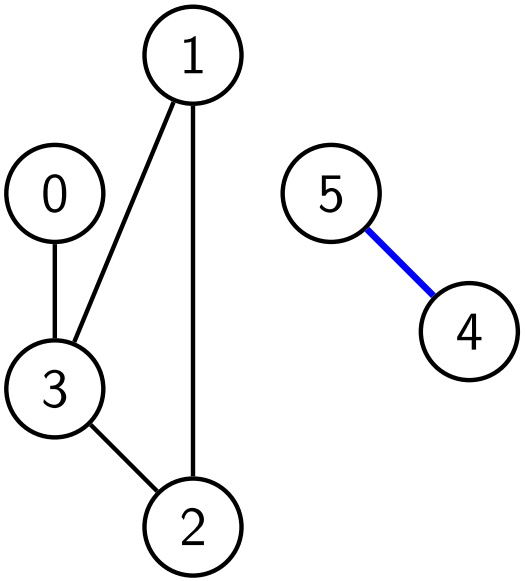
Principales étapes

- Déclaration et initialisatione de la structure de données `union-find`
- Ajouts de tous les sommets de G dans `union-find` (première boucle `for`)
- Traitement des arcs de G et potentiels unions (seconde boucle `for`)

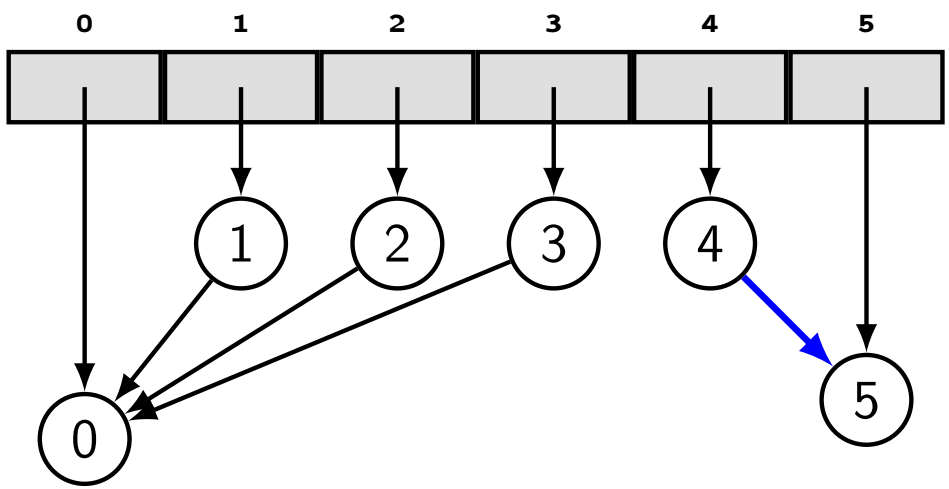
$$(0,3),\ (1,2),\ (1,3),\ (2,3),\ (5,4)$$

Déroulement de l'algorithme

Graphe G



Structure de données `union-find`



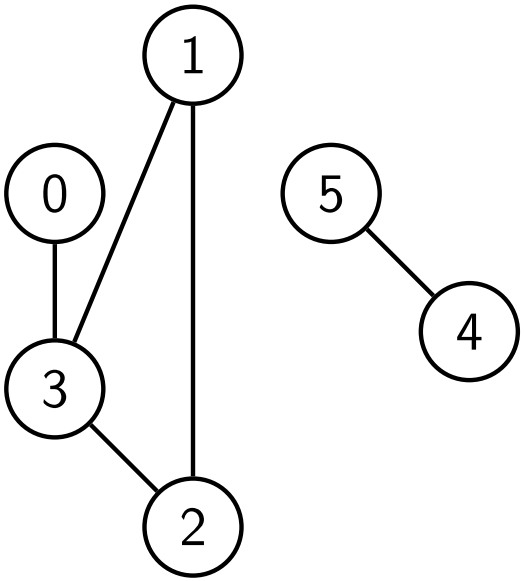
Principales étapes

- Déclaration et initialisatione de la structure de données `union-find`
- Ajouts de tous les sommets de G dans `union-find` (première boucle `for`)
- Traitement des arcs de G et potentiels unions (seconde boucle `for`)

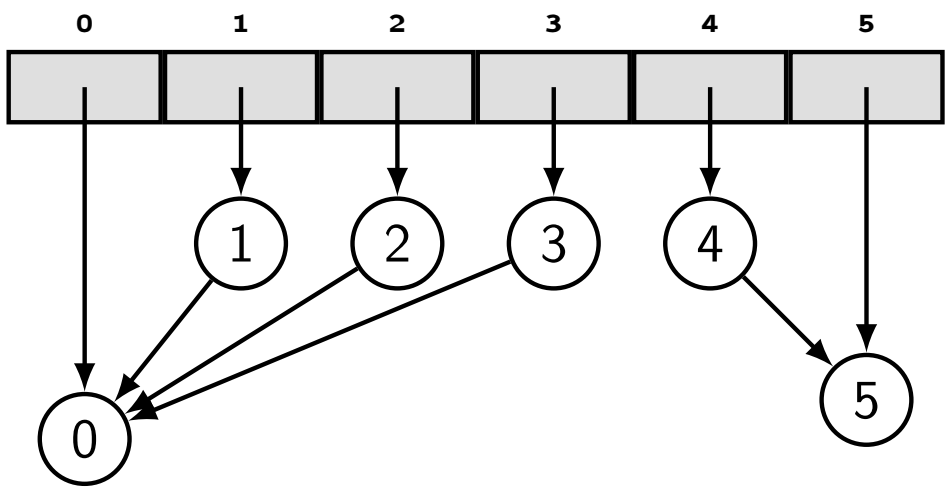
$$(0,3),\ (1,2),\ (1,3),\ (2,3),\ (5,4)$$

Déroulement de l'algorithme

Graphe G



Structure de données `union-find`



Principales étapes

- Déclaration et initialisatione de la structure de données `union-find`
- Ajouts de tous les sommets de G dans `union-find` (première boucle `for`)
- Traitement des arcs de G et potentiels unions (seconde boucle `for`)

$$(0,3),\ (1,2),\ (1,3),\ (2,3),\ (5,4)$$

- A la fin on a bien 2 sous-ensembles disjoints

Calcul des composantes fortement connexes d'un digraphe

Principe : Algorithme de Kosaraju-Sharir

Soit G un digraphe, l'algorithme opère en deux étapes :

- Effectuer un parcours en profondeur de G et enregistrer les dates de fin de visite
- Effectuer un parcours en profondeur sur le graphe transposé G^T de G, en suivant l'ordre décroissant des dates de fin données par la première étape.

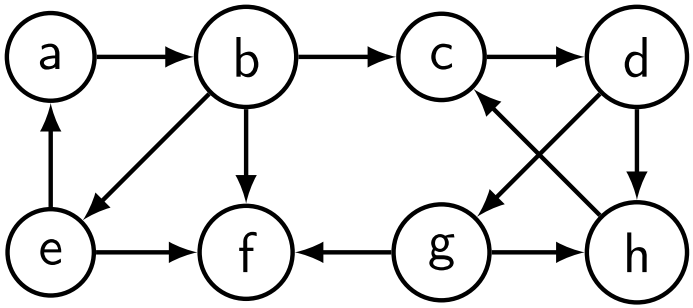
Les arbres produits par le deuxième parcours sont les composantes fortement connexes de G.

Complexité

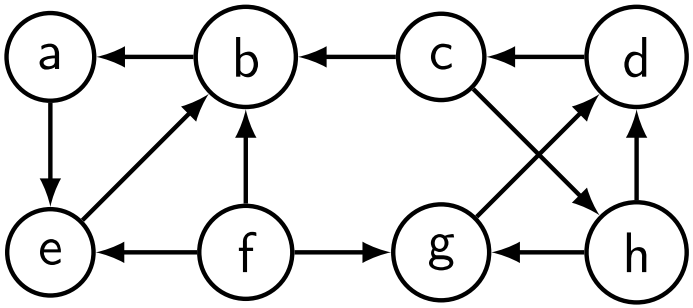
Pour un digraphe $G = (S, A)$, cet algorithme a une complexité linéaire en nombre de sommets et nombre d'arcs, c'est-à-dire, $\mathcal{O}(|S| + |A|)$

Déroulement de l'algorithme

Graphe G



Graphe transposé G^T de G

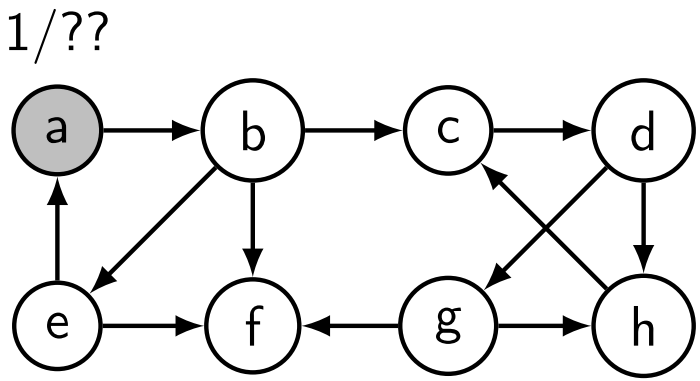


Déroulement de l'algorithme

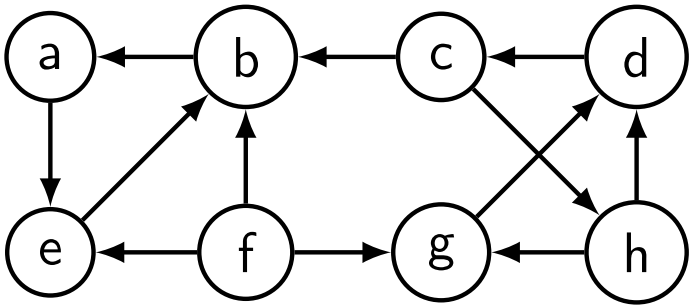
1. Parcours en profondeur de G et tri par ordre décroissant des dates de fin :
2. Parcours en profondeur de G^T en suivant l'ordre des sommets donnés à l'étape 1

Déroulement de l'algorithme

Graphe G



Graphe transposé G^T de G

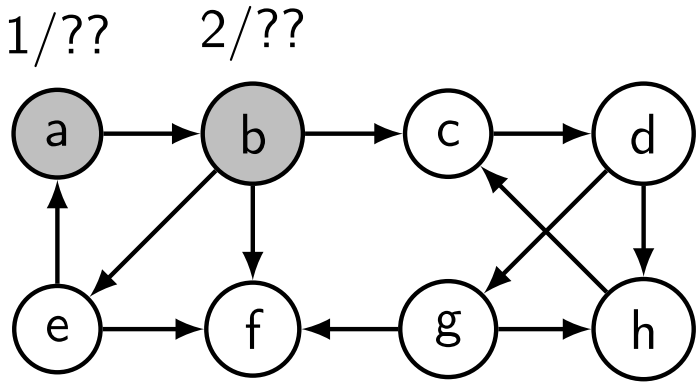


Déroulement de l'algorithme

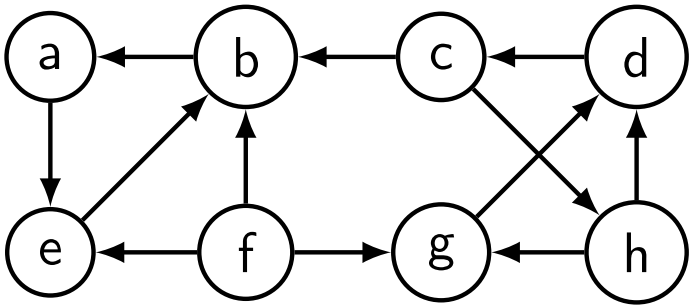
1. Parcours en profondeur de G et tri par ordre décroissant des dates de fin :
2. Parcours en profondeur de G^T en suivant l'ordre des sommets donnés à l'étape 1

Déroulement de l'algorithme

Graphe G



Graphe transposé G^T de G

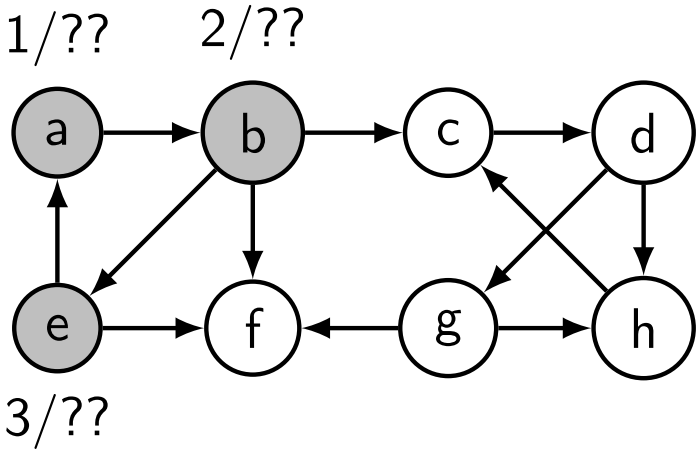


Déroulement de l'algorithme

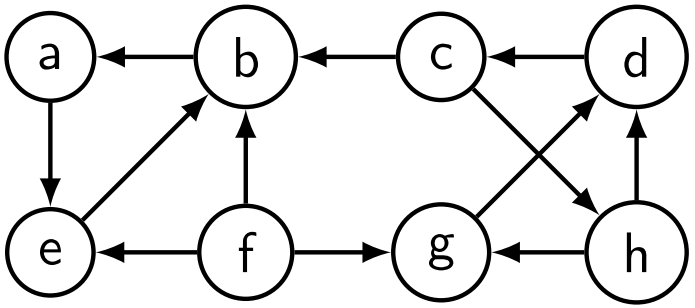
1. Parcours en profondeur de G et tri par ordre décroissant des dates de fin :
2. Parcours en profondeur de G^T en suivant l'ordre des sommets donnés à l'étape 1

Déroulement de l'algorithme

Graphe G



Graphe transposé G^T de G

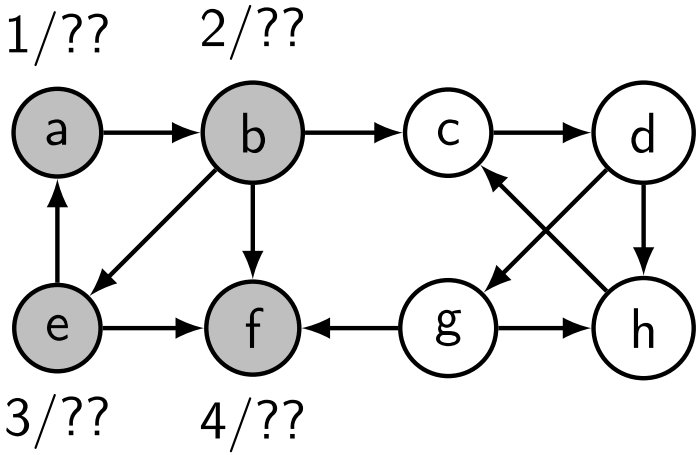


Déroulement de l'algorithme

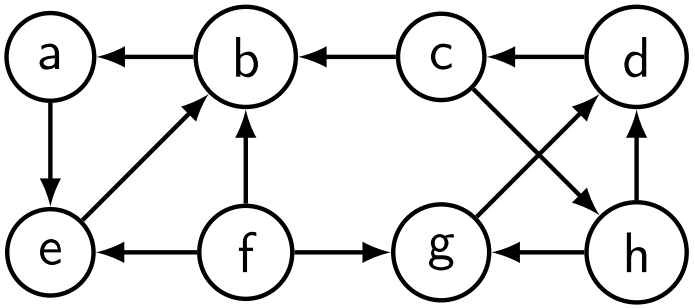
1. Parcours en profondeur de G et tri par ordre décroissant des dates de fin :
2. Parcours en profondeur de G^T en suivant l'ordre des sommets donnés à l'étape 1

Déroulement de l'algorithme

Graphe G



Graphe transposé G^T de G

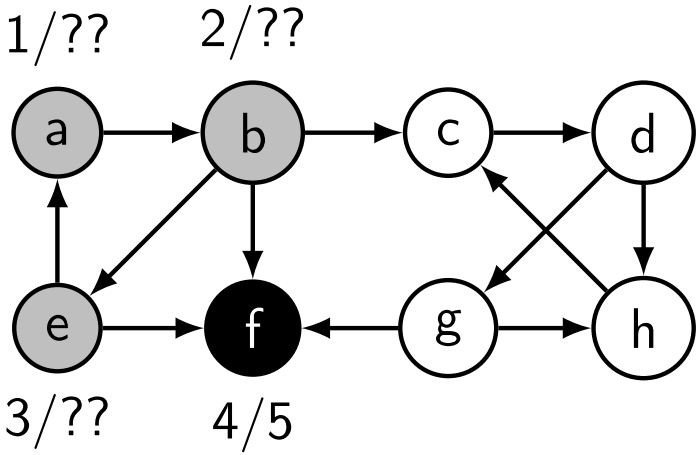


Déroulement de l'algorithme

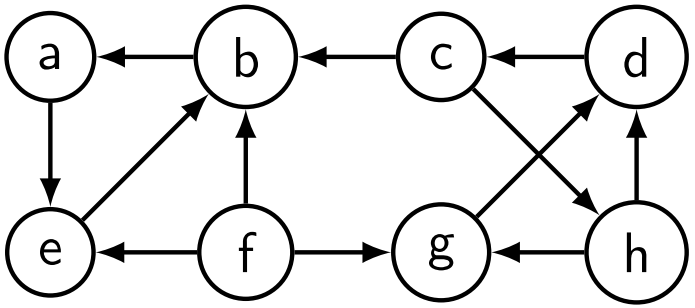
1. Parcours en profondeur de G et tri par ordre décroissant des dates de fin :
2. Parcours en profondeur de G^T en suivant l'ordre des sommets donnés à l'étape 1

Déroulement de l'algorithme

Graphe G



Graphe transposé G^T de G

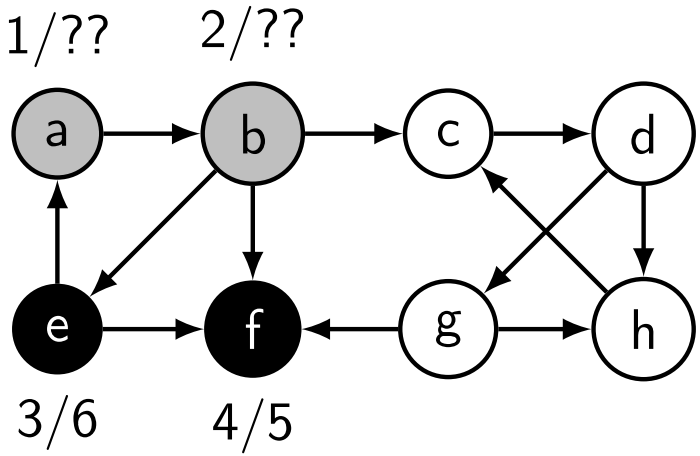


Déroulement de l'algorithme

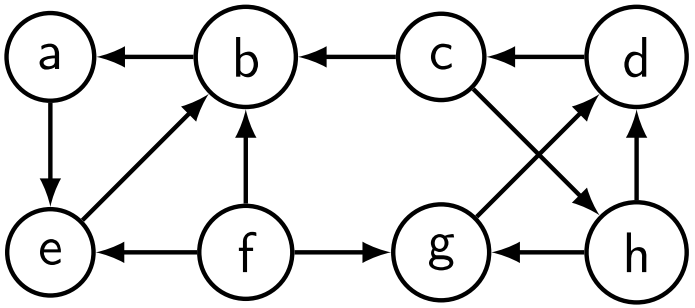
1. Parcours en profondeur de G et tri par ordre décroissant des dates de fin :
2. Parcours en profondeur de G^T en suivant l'ordre des sommets donnés à l'étape 1

Déroulement de l'algorithme

Graphe G



Graphe transposé G^T de G

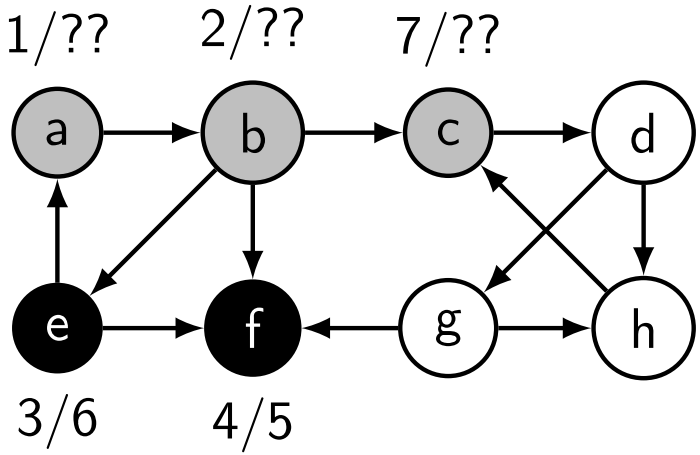


Déroulement de l'algorithme

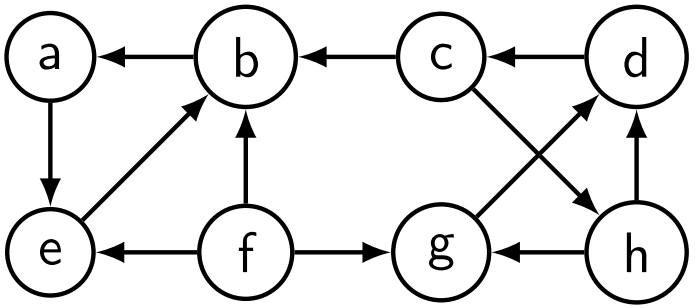
1. Parcours en profondeur de G et tri par ordre décroissant des dates de fin :
2. Parcours en profondeur de G^T en suivant l'ordre des sommets donnés à l'étape 1

Déroulement de l'algorithme

Graphe G



Graphe transposé G^T de G

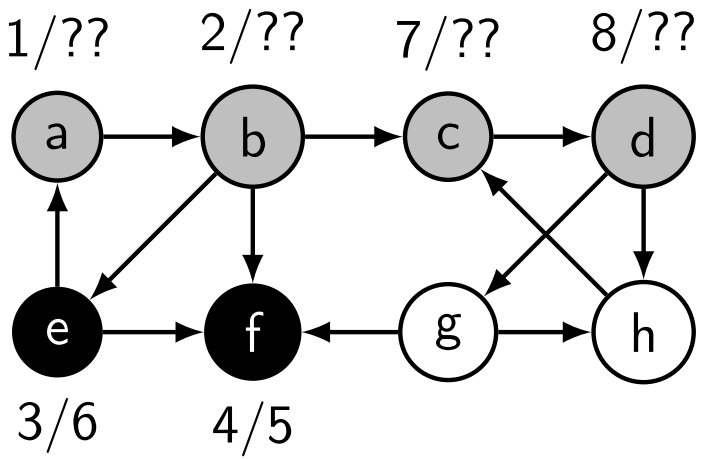


Déroulement de l'algorithme

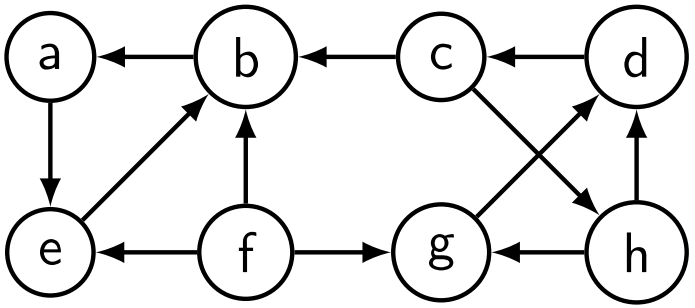
1. Parcours en profondeur de G et tri par ordre décroissant des dates de fin :
2. Parcours en profondeur de G^T en suivant l'ordre des sommets donnés à l'étape 1

Déroulement de l'algorithme

Graphe G



Graphe transposé G^T de G

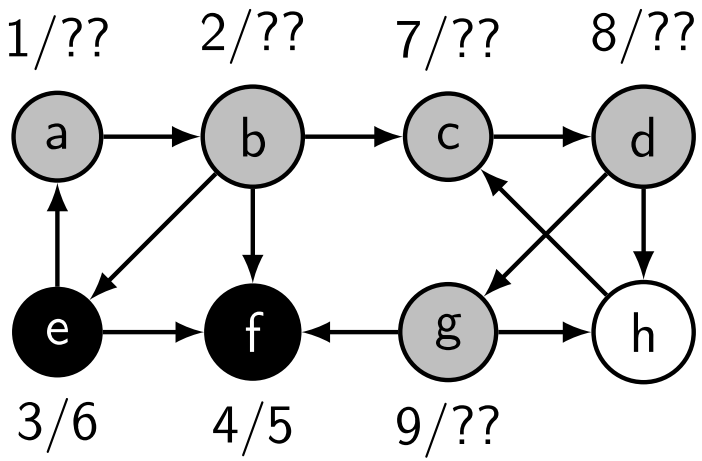


Déroulement de l'algorithme

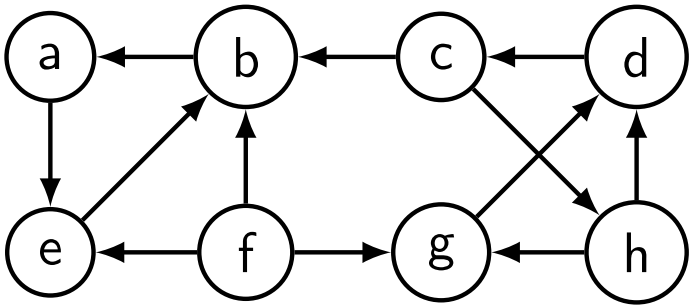
1. Parcours en profondeur de G et tri par ordre décroissant des dates de fin :
2. Parcours en profondeur de G^T en suivant l'ordre des sommets donnés à l'étape 1

Déroulement de l'algorithme

Graphe G



Graphe transposé G^T de G

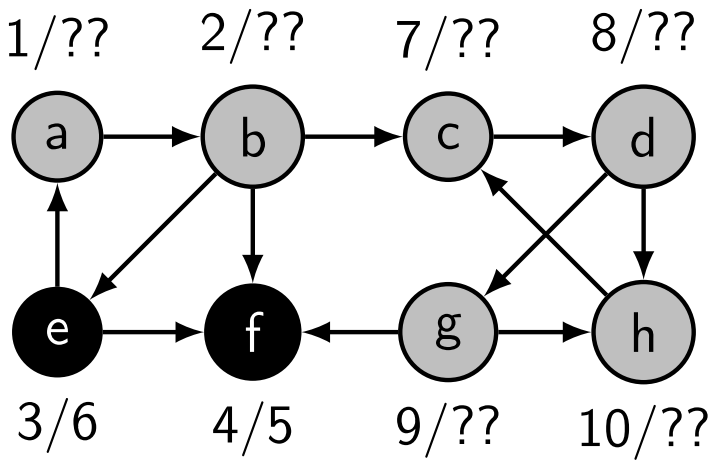


Déroulement de l'algorithme

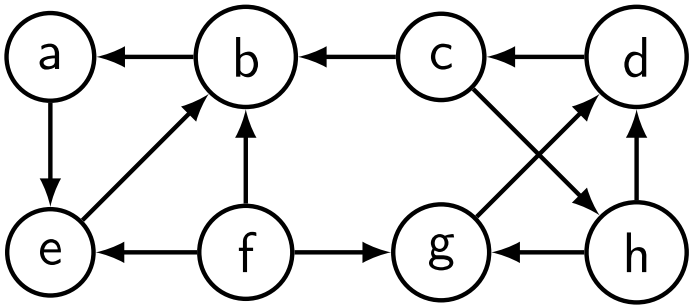
1. Parcours en profondeur de G et tri par ordre décroissant des dates de fin :
2. Parcours en profondeur de G^T en suivant l'ordre des sommets donnés à l'étape 1

Déroulement de l'algorithme

Graphe G



Graphe transposé G^T de G

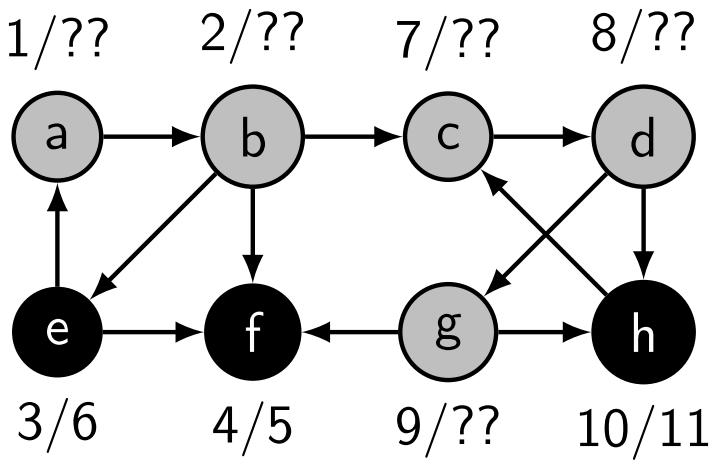


Déroulement de l'algorithme

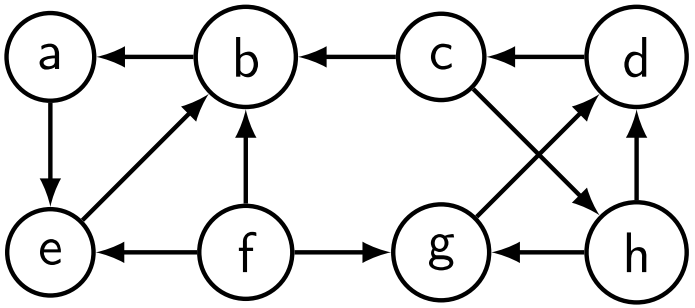
1. Parcours en profondeur de G et tri par ordre décroissant des dates de fin :
2. Parcours en profondeur de G^T en suivant l'ordre des sommets donnés à l'étape 1

Déroulement de l'algorithme

Graphe G



Graphe transposé G^T de G

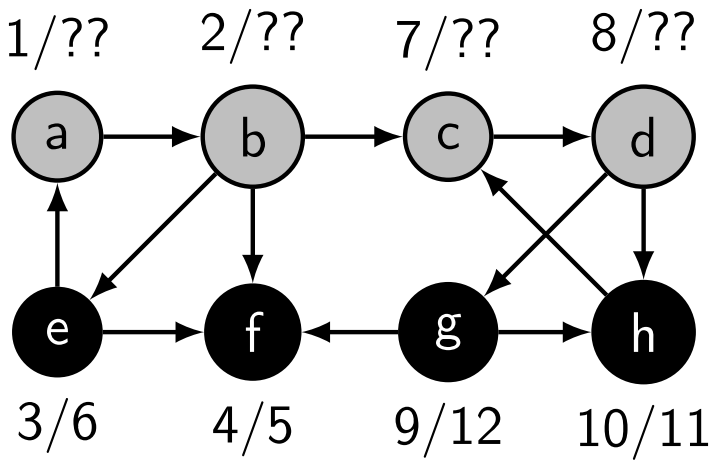


Déroulement de l'algorithme

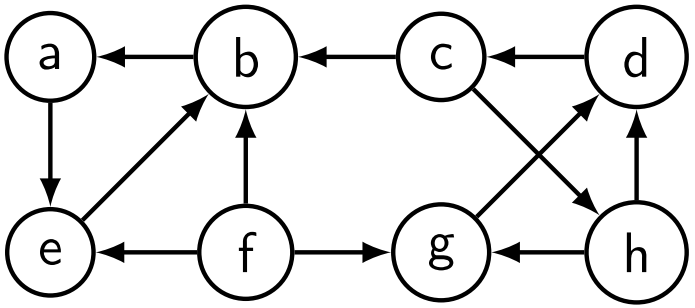
1. Parcours en profondeur de G et tri par ordre décroissant des dates de fin :
2. Parcours en profondeur de G^T en suivant l'ordre des sommets donnés à l'étape 1

Déroulement de l'algorithme

Graphe G



Graphe transposé G^T de G

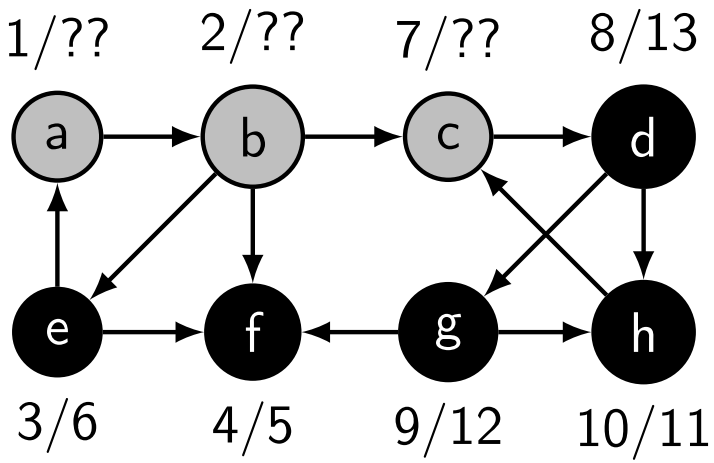


Déroulement de l'algorithme

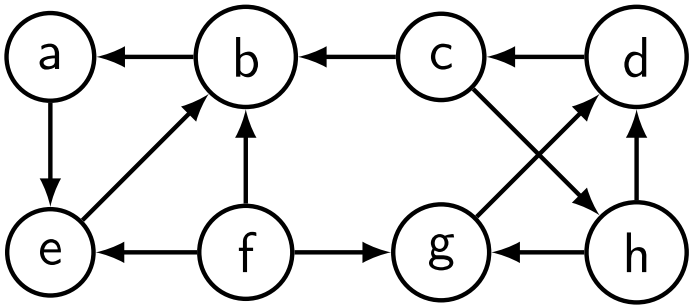
1. Parcours en profondeur de G et tri par ordre décroissant des dates de fin :
2. Parcours en profondeur de G^T en suivant l'ordre des sommets donnés à l'étape 1

Déroulement de l'algorithme

Graphe G



Graphe transposé G^T de G

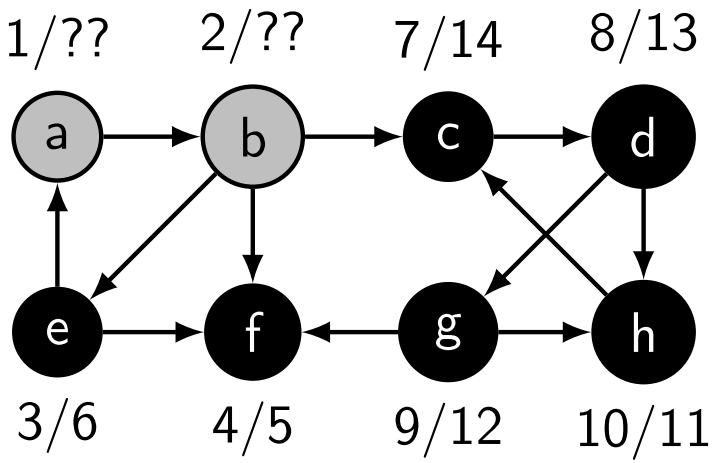


Déroulement de l'algorithme

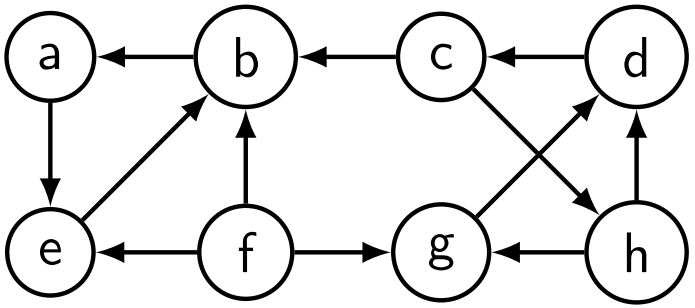
1. Parcours en profondeur de G et tri par ordre décroissant des dates de fin :
2. Parcours en profondeur de G^T en suivant l'ordre des sommets donnés à l'étape 1

Déroulement de l'algorithme

Graphe G



Graphe transposé G^T de G

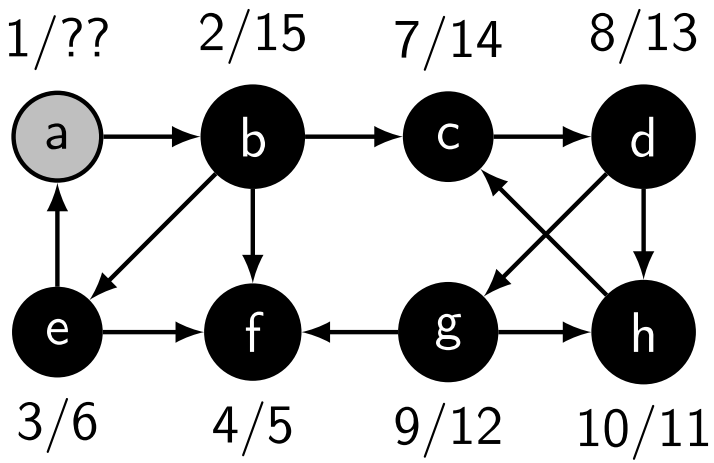


Déroulement de l'algorithme

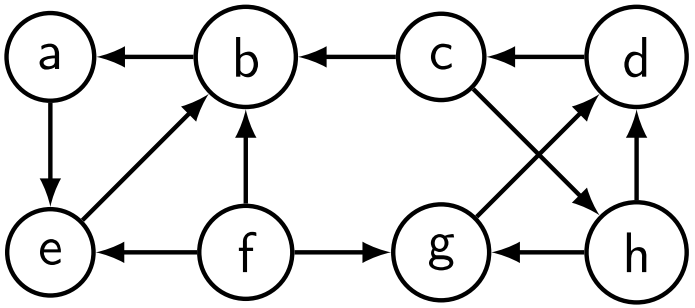
1. Parcours en profondeur de G et tri par ordre décroissant des dates de fin :
2. Parcours en profondeur de G^T en suivant l'ordre des sommets donnés à l'étape 1

Déroulement de l'algorithme

Graphe G



Graphe transposé G^T de G

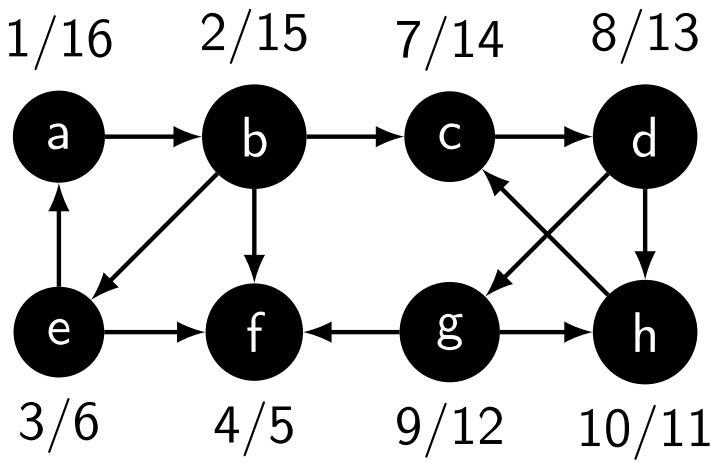


Déroulement de l'algorithme

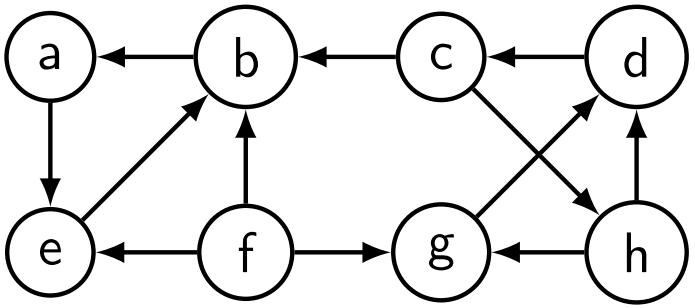
1. Parcours en profondeur de G et tri par ordre décroissant des dates de fin :
2. Parcours en profondeur de G^T en suivant l'ordre des sommets donnés à l'étape 1

Déroulement de l'algorithme

Graphe G



Graphe transposé G^T de G

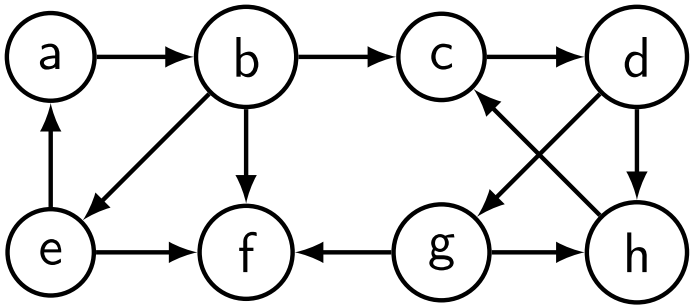


Déroulement de l'algorithme

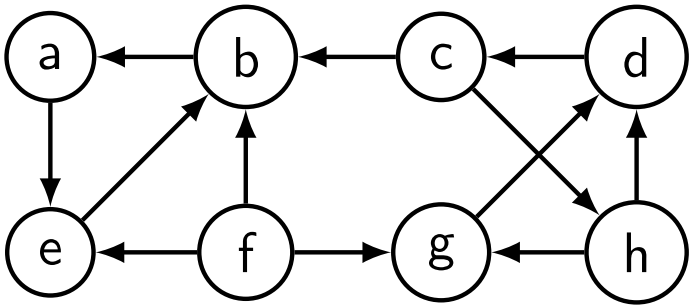
1. Parcours en profondeur de G et tri par ordre décroissant des dates de fin :
2. Parcours en profondeur de G^T en suivant l'ordre des sommets donnés à l'étape 1

Déroulement de l'algorithme

Graphe G



Graphe transposé G^T de G



Déroulement de l'algorithme

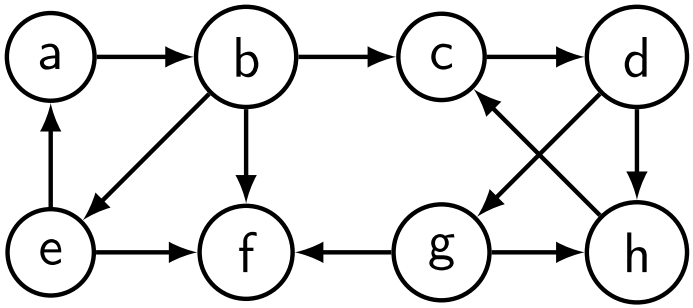
1. Parcours en profondeur de G et tri par ordre décroissant des dates de fin :

   a, b, c, d, g, h, e, f

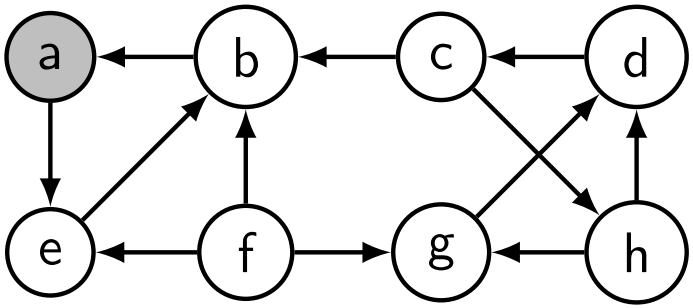
2. Parcours en profondeur de G^T en suivant l'ordre des sommets donnés à l'étape 1

Déroulement de l'algorithme

Graphe G



Graphe transposé G^T de G



Déroulement de l'algorithme

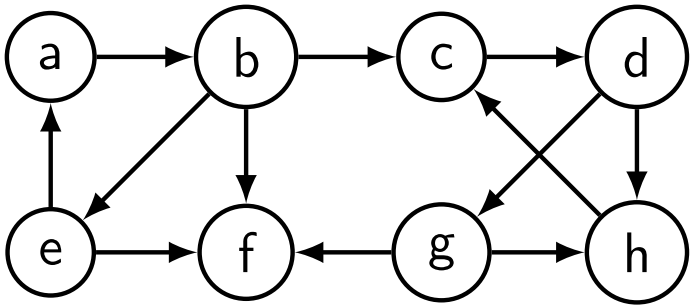
1. Parcours en profondeur de G et tri par ordre décroissant des dates de fin :

   a, b, c, d, g, h, e, f

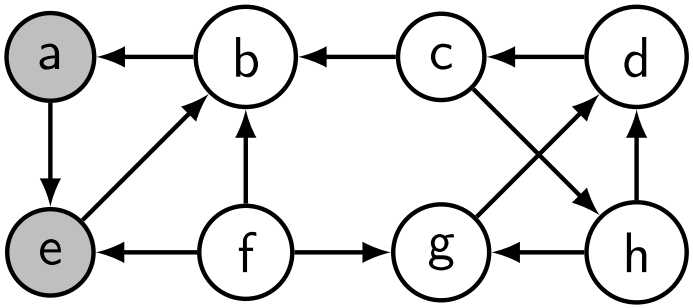
2. Parcours en profondeur de G^T en suivant l'ordre des sommets donnés à l'étape 1

Déroulement de l'algorithme

Graphe G



Graphe transposé G^T de G



Déroulement de l'algorithme

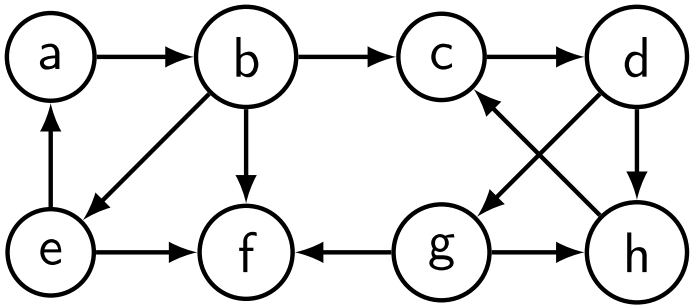
1. Parcours en profondeur de G et tri par ordre décroissant des dates de fin :

   a, b, c, d, g, h, e, f

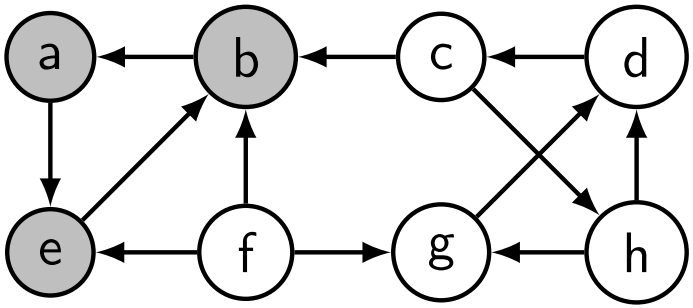
2. Parcours en profondeur de G^T en suivant l'ordre des sommets donnés à l'étape 1

Déroulement de l'algorithme

Graphe G



Graphe transposé G^T de G



Déroulement de l'algorithme

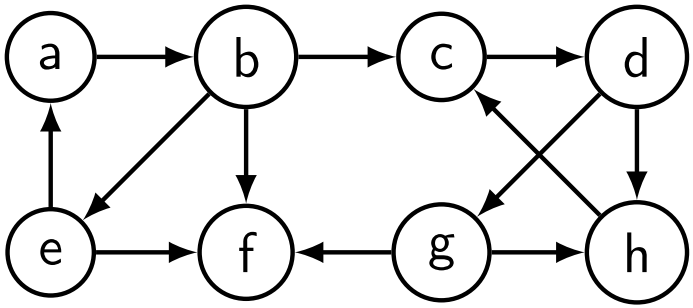
1. Parcours en profondeur de G et tri par ordre décroissant des dates de fin :

   a, b, c, d, g, h, e, f

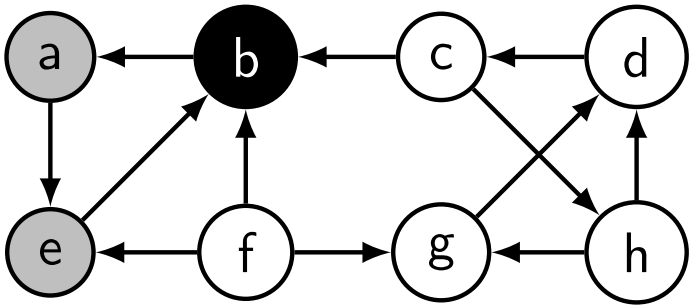
2. Parcours en profondeur de G^T en suivant l'ordre des sommets donnés à l'étape 1

Déroulement de l'algorithme

Graphe G



Graphe transposé G^T de G



Déroulement de l'algorithme

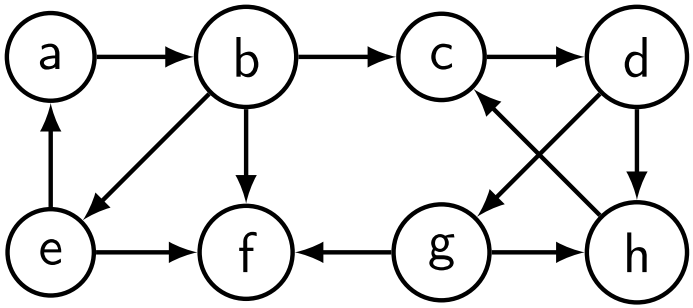
1. Parcours en profondeur de G et tri par ordre décroissant des dates de fin :

   a, b, c, d, g, h, e, f

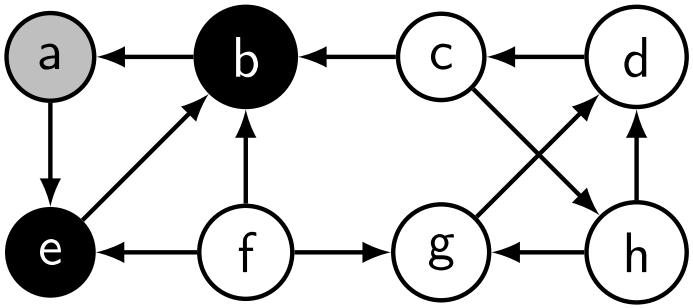
2. Parcours en profondeur de G^T en suivant l'ordre des sommets donnés à l'étape 1

Déroulement de l'algorithme

Graphe G



Graphe transposé G^T de G



Déroulement de l'algorithme

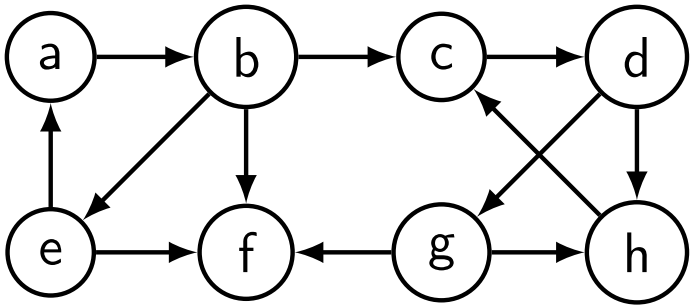
1. Parcours en profondeur de G et tri par ordre décroissant des dates de fin :

   a, b, c, d, g, h, e, f

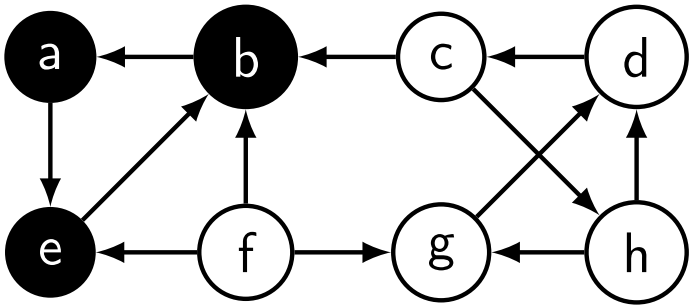
2. Parcours en profondeur de G^T en suivant l'ordre des sommets donnés à l'étape 1

Déroulement de l'algorithme

Graphe G



Graphe transposé G^T de G



Déroulement de l'algorithme

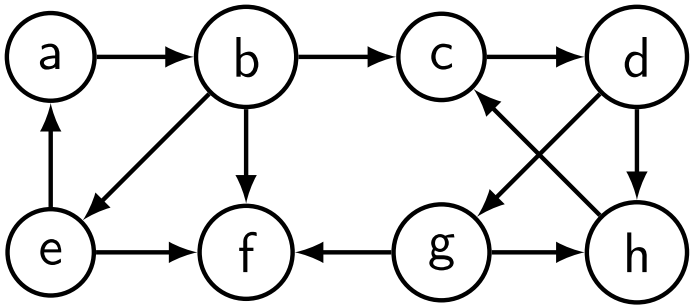
1. Parcours en profondeur de G et tri par ordre décroissant des dates de fin :

   a, b, c, d, g, h, e, f

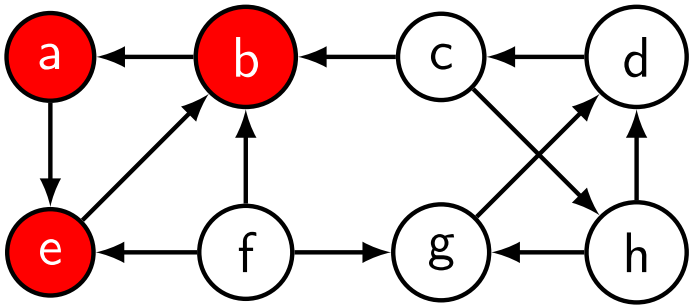
2. Parcours en profondeur de G^T en suivant l'ordre des sommets donnés à l'étape 1

Déroulement de l'algorithme

Graphe G



Graphe transposé G^T de G



Déroulement de l'algorithme

1. Parcours en profondeur de G et tri par ordre décroissant des dates de fin :

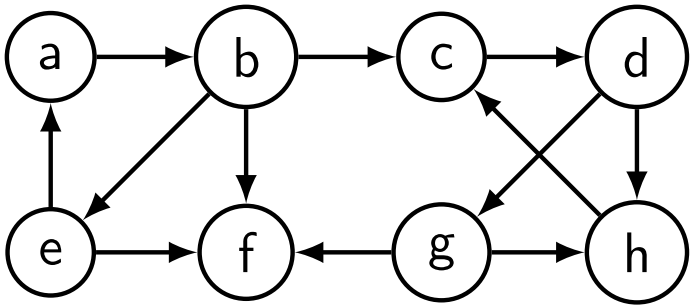
   a, b, c, d, g, h, e, f

2. Parcours en profondeur de G^T en suivant l'ordre des sommets donnés à l'étape 1

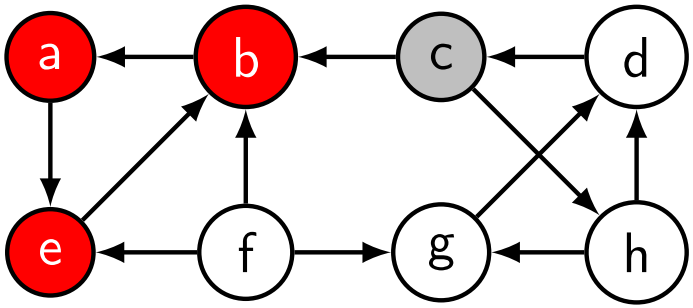
   $$\text{CFC}_1 = \{a, e, b\}$$

Déroulement de l'algorithme

Graphe G



Graphe transposé G^T de G



Déroulement de l'algorithme

1. Parcours en profondeur de G et tri par ordre décroissant des dates de fin :

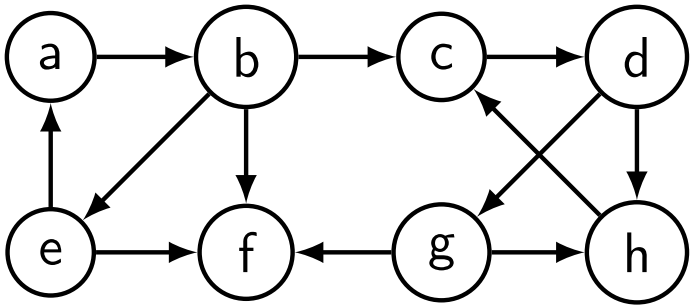
   a, b, c, d, g, h, e, f

2. Parcours en profondeur de G^T en suivant l'ordre des sommets donnés à l'étape 1

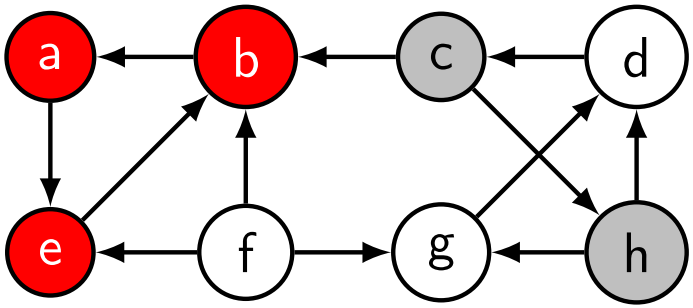
   $$\text{CFC}_1 = \{a, e, b\}$$

Déroulement de l'algorithme

Graphe G



Graphe transposé G^T de G



Déroulement de l'algorithme

1. Parcours en profondeur de G et tri par ordre décroissant des dates de fin :

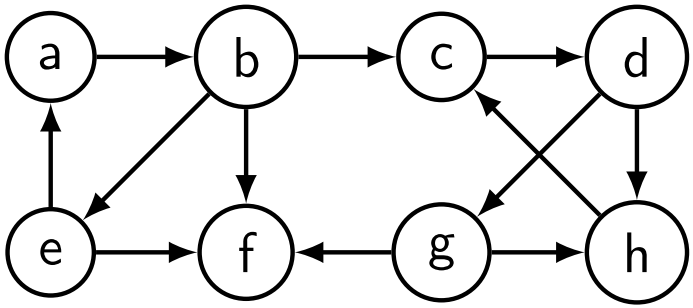
   a, b, c, d, g, h, e, f

2. Parcours en profondeur de G^T en suivant l'ordre des sommets donnés à l'étape 1

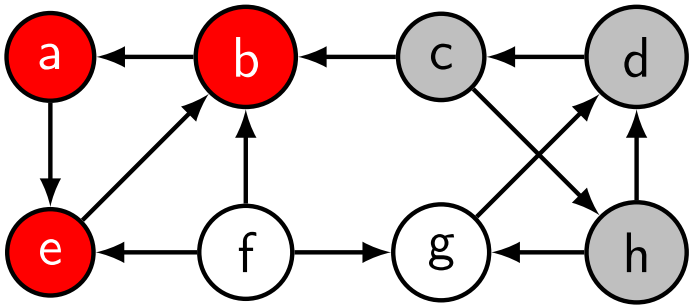
   $$\text{CFC}_1 = \{a, e, b\}$$

Déroulement de l'algorithme

Graphe G



Graphe transposé G^T de G



Déroulement de l'algorithme

1. Parcours en profondeur de G et tri par ordre décroissant des dates de fin :

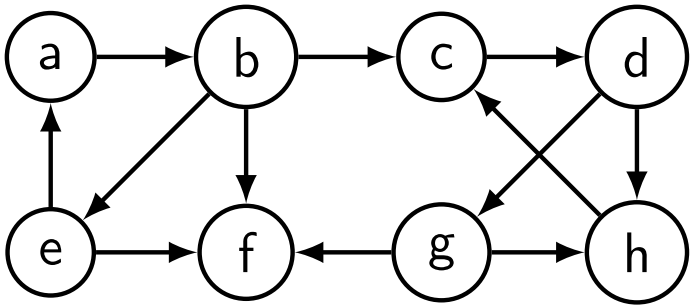
   a, b, c, d, g, h, e, f

2. Parcours en profondeur de G^T en suivant l'ordre des sommets donnés à l'étape 1

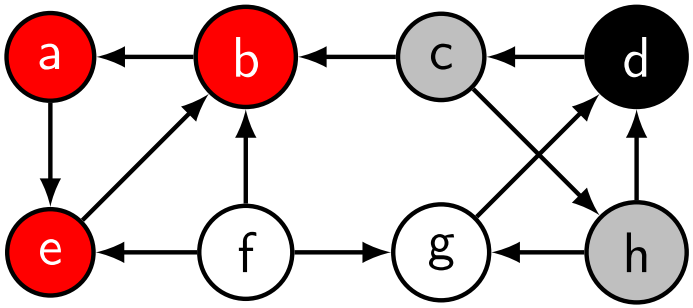
   $$\text{CFC}_1 = \{a, e, b\}$$

Déroulement de l'algorithme

Graphe G



Graphe transposé G^T de G



Déroulement de l'algorithme

1. Parcours en profondeur de G et tri par ordre décroissant des dates de fin :

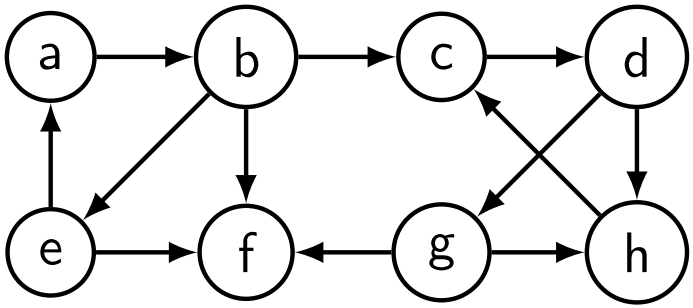
   a, b, c, d, g, h, e, f

2. Parcours en profondeur de G^T en suivant l'ordre des sommets donnés à l'étape 1

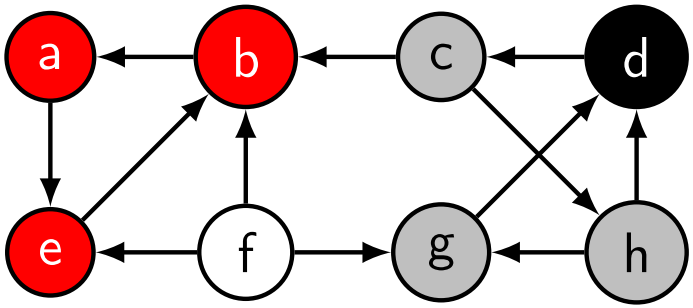
   $$\text{CFC}_1 = \{a, e, b\}$$

Déroulement de l'algorithme

Graphe G



Graphe transposé G^T de G



Déroulement de l'algorithme

1. Parcours en profondeur de G et tri par ordre décroissant des dates de fin :

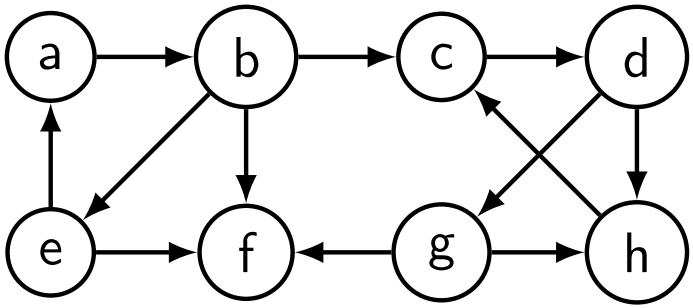
   a, b, c, d, g, h, e, f

2. Parcours en profondeur de G^T en suivant l'ordre des sommets donnés à l'étape 1

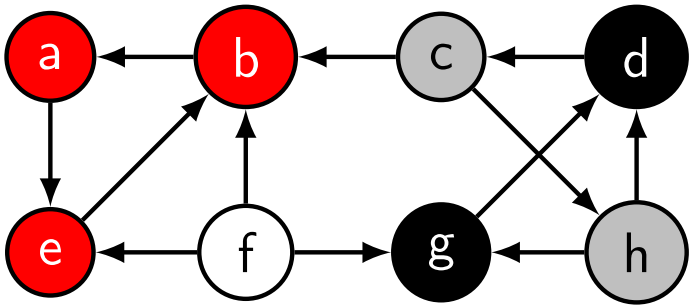
   $$\text{CFC}_1 = \{a, e, b\}$$

Déroulement de l'algorithme

Graphe G



Graphe transposé G^T de G



Déroulement de l'algorithme

1. Parcours en profondeur de G et tri par ordre décroissant des dates de fin :

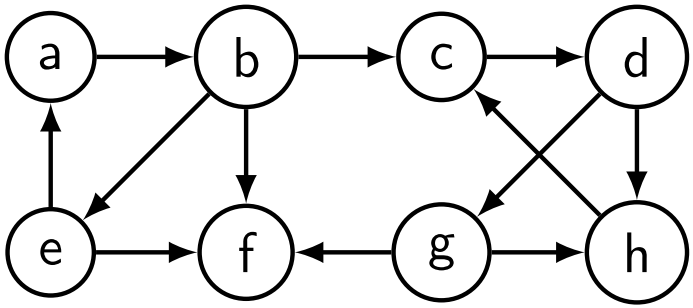
   a, b, c, d, g, h, e, f

2. Parcours en profondeur de G^T en suivant l'ordre des sommets donnés à l'étape 1

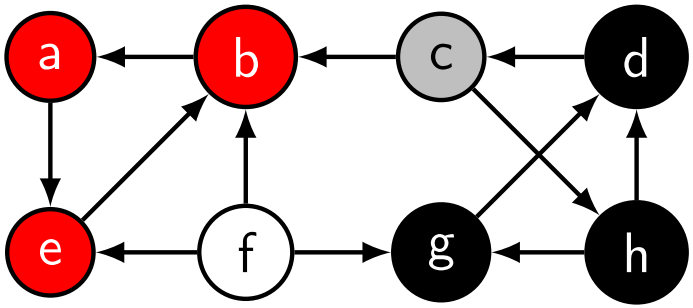
   $$\text{CFC}_1 = \{a, e, b\}$$

Déroulement de l'algorithme

Graphe G



Graphe transposé G^T de G



Déroulement de l'algorithme

1. Parcours en profondeur de G et tri par ordre décroissant des dates de fin :

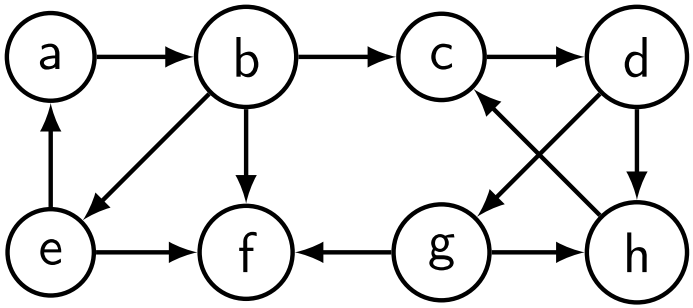
   a, b, c, d, g, h, e, f

2. Parcours en profondeur de G^T en suivant l'ordre des sommets donnés à l'étape 1

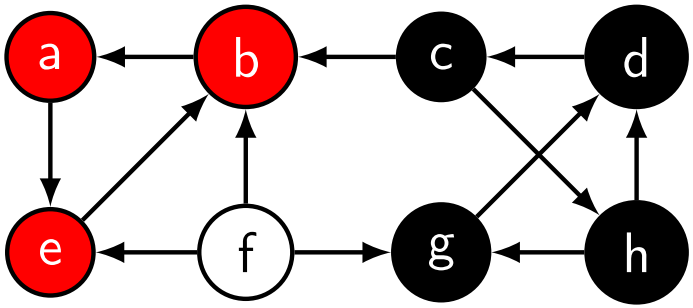
   $\text{CFC}_1 = \{a, e, b\}$

Déroulement de l'algorithme

Graphe G



Graphe transposé G^T de G



Déroulement de l'algorithme

1. Parcours en profondeur de G et tri par ordre décroissant des dates de fin :

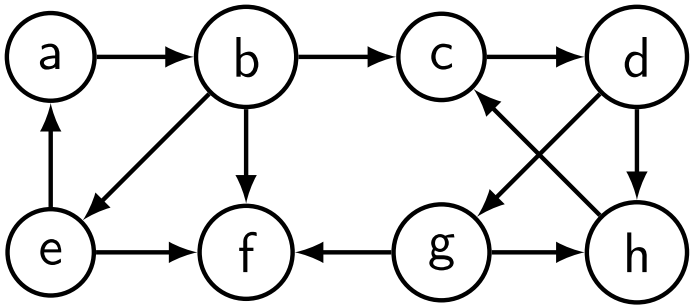
   a, b, c, d, g, h, e, f

2. Parcours en profondeur de G^T en suivant l'ordre des sommets donnés à l'étape 1

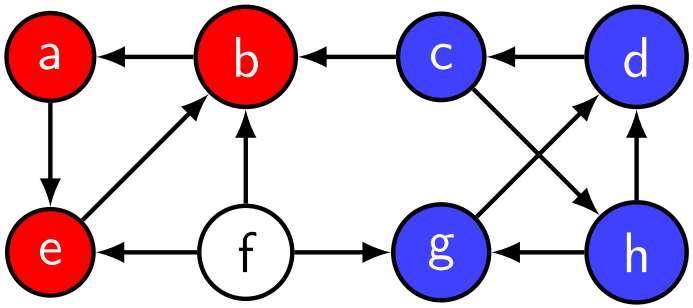
   $$\text{CFC}_1 = \{a, e, b\}$$

Déroulement de l'algorithme

Graphe G



Graphe transposé G^T de G



Déroulement de l'algorithme

1. Parcours en profondeur de G et tri par ordre décroissant des dates de fin :

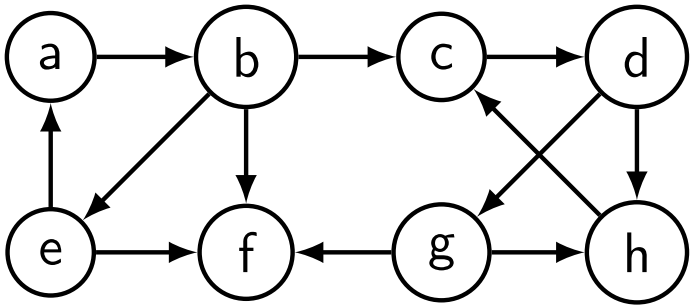
   a, b, c, d, g, h, e, f

2. Parcours en profondeur de G^T en suivant l'ordre des sommets donnés à l'étape 1

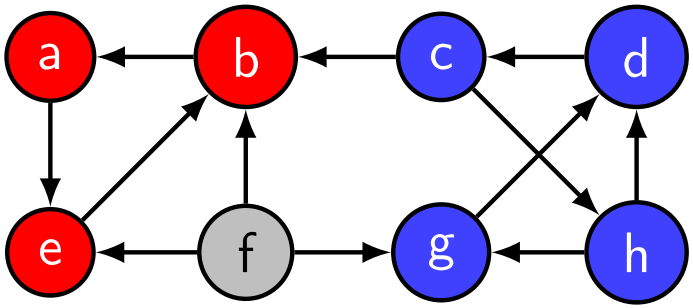
   $\text{CFC}_1 = \{a, e, b\}$, $\text{CFC}_2 = \{c, h, d, g\}$

Déroulement de l'algorithme

Graphe G



Graphe transposé G^T de G



Déroulement de l'algorithme

1. Parcours en profondeur de G et tri par ordre décroissant des dates de fin :

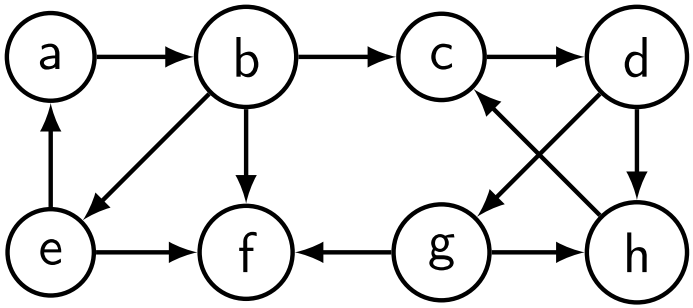
   a, b, c, d, g, h, e, f

2. Parcours en profondeur de G^T en suivant l'ordre des sommets donnés à l'étape 1

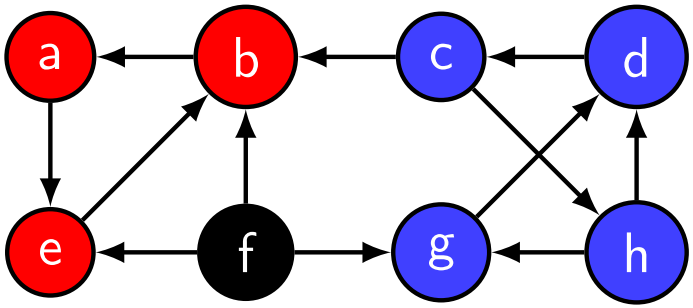
   $$\text{CFC}_1 = \{a, e, b\},\ \text{CFC}_2 = \{c, h, d, g\}$$

Déroulement de l'algorithme

Graphe G



Graphe transposé G^T de G



Déroulement de l'algorithme

1. Parcours en profondeur de G et tri par ordre décroissant des dates de fin :

   a, b, c, d, g, h, e, f

2. Parcours en profondeur de G^T en suivant l'ordre des sommets donnés à l'étape 1

$$\text{CFC}_1 = \{a, e, b\},\ \text{CFC}_2 = \{c, h, d, g\}\ ,\ \text{CFC}_3 = \{f\}$$

Quelques éléments de correction

Pour plus d'explications *cf* Cormen *et al.*, chapitre 22.5, les principaux résultats

- G et G^T ont les mêmes composantes fortement connexes.
- Lemme 1 : Soient C et C' des CFC distinctes de $G = (S, A)$, soit $u, v \in C$, soit $u', v' \in C'$, et supposons qu'il y ait un chemin u vers u' dans G. Alors, il ne peut pas y avoir aussi un chemin v' vers v dans G.
- Lemme 2 Soient C et C' des CFC distinctes de $G = (S, A)$. On suppose qu'il y a un arc $(u, v) \in A$, tel que $u \in C$ et $v \in C'$. Alors, $f(C) > f(C')$.
- Corollaire Soient C et C' des CFC distinctes de $G = (S, A)$. Supposons qu'il y ait un arc $(u, v) \in^T A$, tel que $u \in C$ et $v \in C'$. Alors, $f(C) < f(C')$.

Remarque

L'algorithme de Tarjan est un autre algorithme de calcul des CFC de complexité linaire mais qui réalise qu'un seul parcours en profondeur.

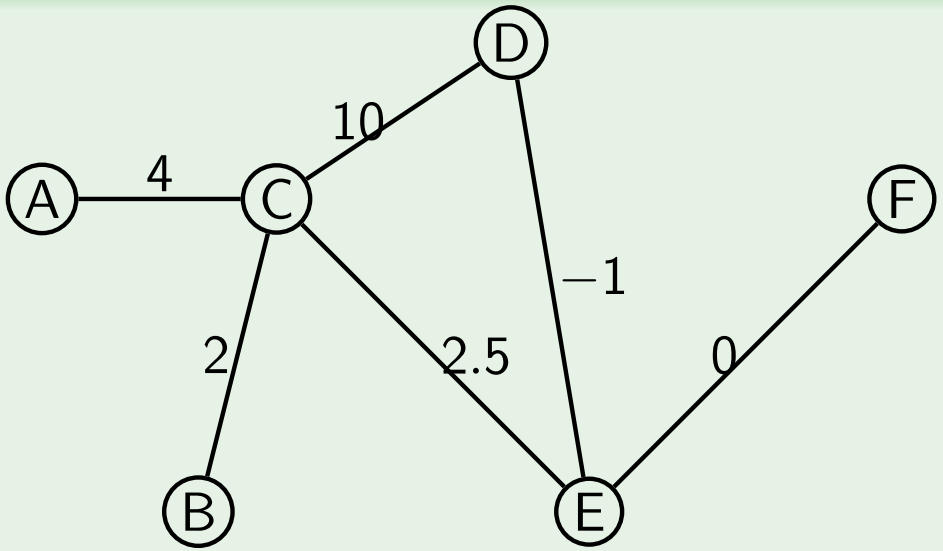
(di)graphes pondérés

Graphes pondérés

Un graphe non orienté (ou simplement graphe) pondéré est un triplet (S, A, w)

- S ensemble fini de sommets (*vertex/vertices*)
- A ensemble fini d'arêtes (*edges*) sous ensemble de $S \times S$
- w une fonction de $A \times A \mapsto \mathbb{R}$ (ou $\mathbb{Z}$ ou autre) qui associe à chaque arête un poids, une capacité ou une valuation.

Exemple



$$w(a, c) = 4$$
$$w(c, b) = 2$$
$$w(c, d) = 10$$
$$w(c, e) = 2.5$$
$$w(d, e) = -1$$
$$w(e, f) = 0$$

Mise en œuvre dans la bibliothèque `libin103`

Définition d'un graphe avec sommets identifiés avec des entiers

```
typedef struct integer_adjlist_elmt_ {
  int vertex;
  double weight;
} integer_adjlist_elmt_t;

typedef struct integer_adjlist_ {
  int vertex;
  generic_set_t adjacent;
} integer_adjlist_t;

typedef struct integer_graph_ {
  int vcount;
  int ecount;
  generic_list_t adjlists;
} integer_graph_t;
```

Point de vigilance !

Une liste d'adjacence est modélisée par

- une liste chaînée (générique) de structures
- dont un des champs est un ensemble (générique)
- dont les éléments sont des structures.

Remarque on considère des (di)graphes dont les arcs/arêtes peuvent avoir une valeur (`weight`).

API des graphes d'entiers

API insertion d'arête/arc

```
int integer_graph_ins_edge(integer_graph_t *g,
                           int v1, int v2, double w);
```

API bonus : liste des voisins d'un sommet (sans les poids !)

```
integer_list_t* integer_graph_adjlist (integer_graph_t* graph, int vertex);
```

Réécriture du parcours en largeur : boucle principale

```
while (integer_queue_size (\&queue) > 0) {
  int vertex; integer_queue_dequeue (\&queue, \&vertex);
  printf ("Vertex %d visited\n", vertex);

  integer_list_t* neighbors = integer_graph_adjlist (graph, vertex);

  integer_list_elmt_t* elem = integer_list_head (neighbors);
  for (; elem != NULL; elem = integer_list_next (elem)) {
    int n = integer_list_data (elem);
    if (!integer_set_is_member (\&set, n)) {
      integer_set_insert (\&set, n);
      integer_queue_enqueue (\&queue, n);
    }
  }
  free(neighbors);
}
```

API des parcours les graphes d'entiers

API parcours en profondeur

```
int integer_dfs(integer_graph_t *graph, int start,
                integer_list_t *ordered);

int integer_dfs_all(integer_graph_t *graph,
                    integer_list_t *ordered);
```

API parcours en largeur

```
int integer_bfs(integer_graph_t *graph, int start,
                integer_list_t *ordered);

int integer_bfs_all(integer_graph_t *graph,
                    integer_list_t *ordered);
```

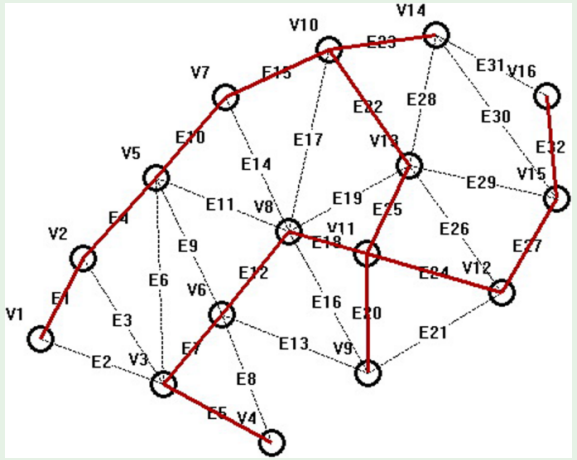
API tri topologique

```
int integer_ts (integer_graph_t* graph, integer_list_t* ordered);
```

Arbre couvrant de poids minimum

Introduction au problème

Exemple de problèmes



Différents algorithmes connus

- Algorithme de Kruskal (considéré dans ce cours) qui utilise
  - une structure de données `tas min`
  - une structure de données `union-find`

Complexité temporelle

Classe de complexité linéaire c'est-à-dire $\mathcal{O}(|A| \log(|S|))$ [1]

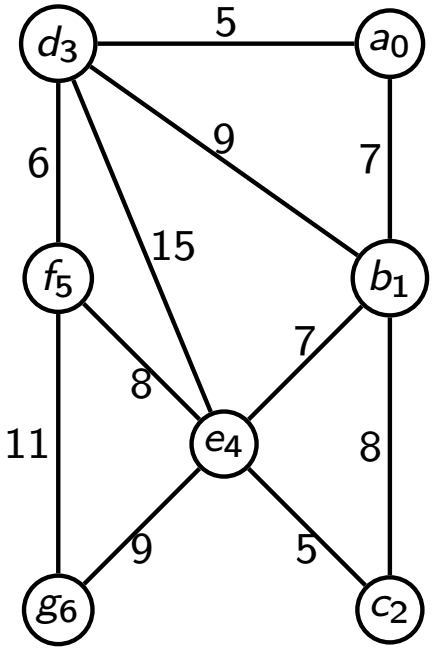
1. *cf* RO201 en 2A Math ou RO202 en 2A Info pour plus de détails

Algorithme de Kruskal

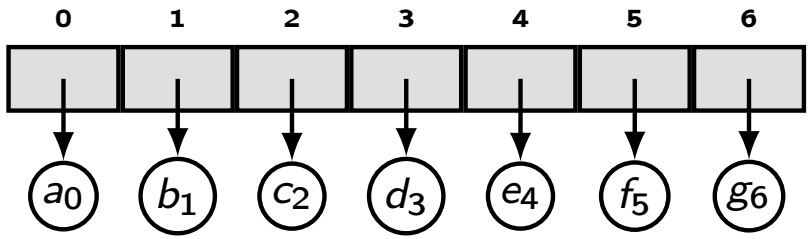
Pseudo-code

```
Function kruskal(graph G=(S,A)) :
    uf_init (&dset, |S|);
    heap_init (&heap_min);
    for /* Tous les sommets s de S */ do
        uf_make_set (&dset, s);
    end
    /* Trier les aretes dans l'ordre croissant des poids */
    for /* Toutes les aretes a de A */ do
        heap_insert (&heap_min, a);
    end
    while heap_size (&heap_min) > 0 do
        (u,v) = heap_extract (&heap_min);
        if not uf_are_connected(&dset, u, v) then
            list_ins_next (&list, NULL, (u,v));
            uf_union (&dset, u, v);
        end
    end
```

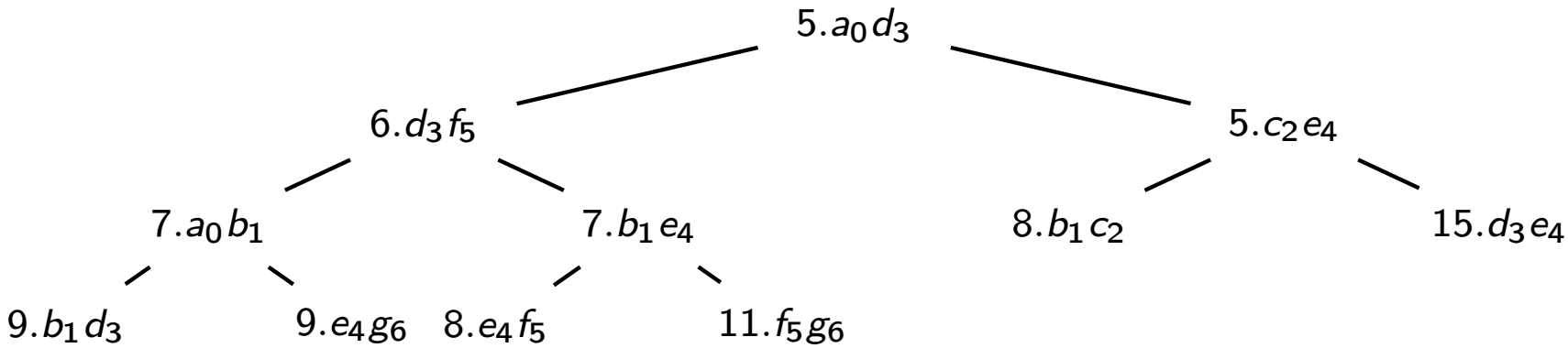
Déroulement de l'algorithme - après les deux boucles for



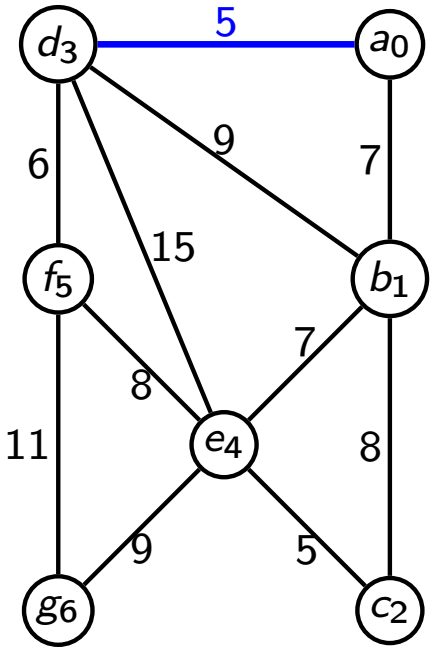
`union-find` des sommets



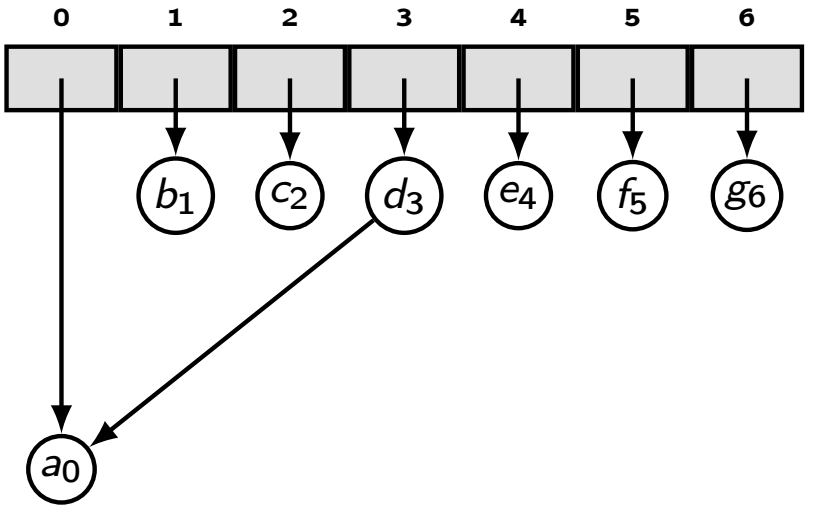
Tas min des arêtes



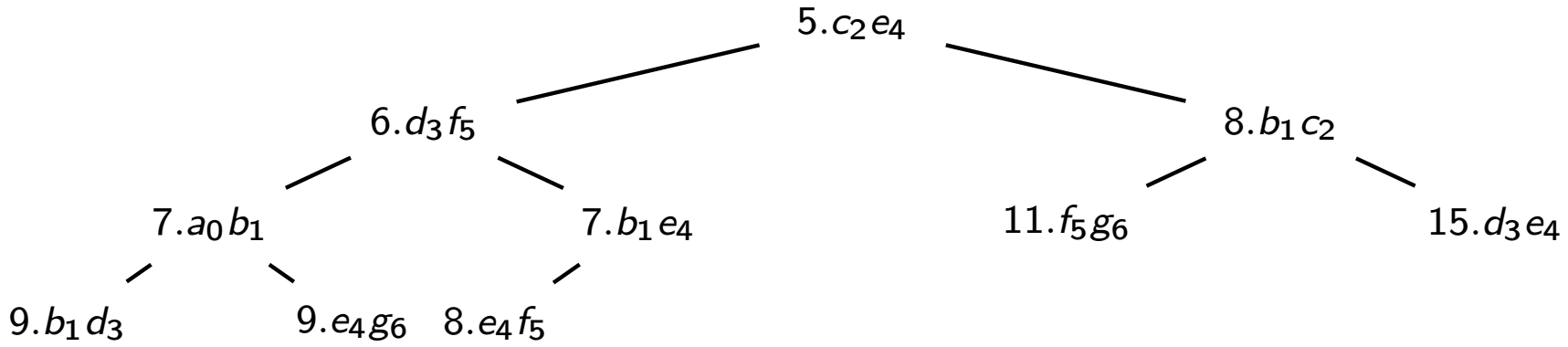
Déroulement de l'algorithme - boucle while



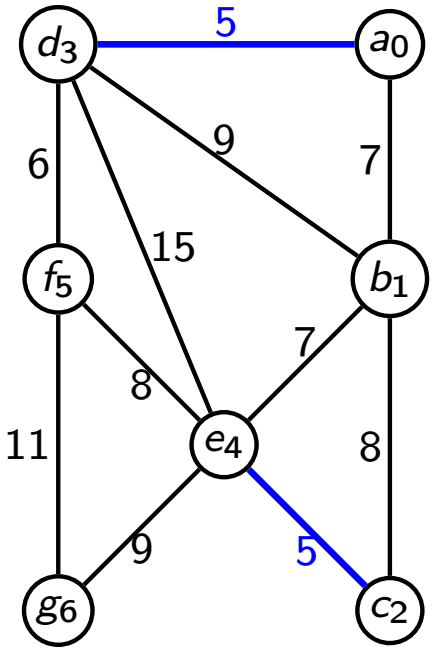
union-find des sommets



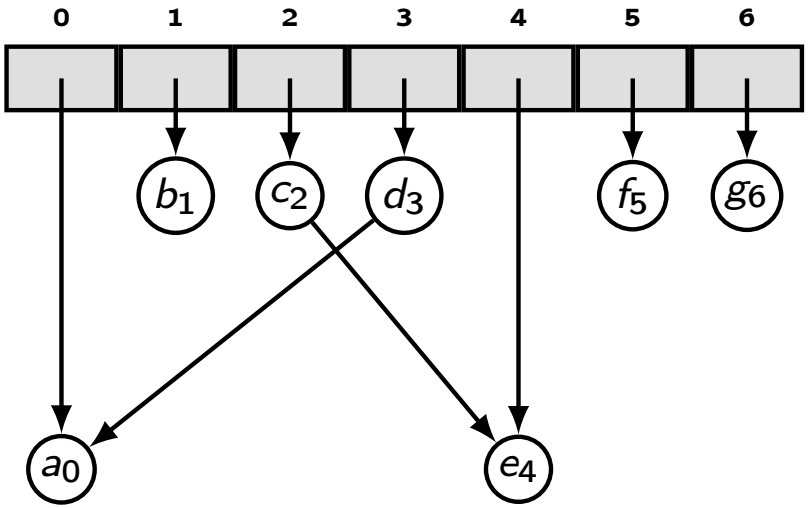
Tas min des arêtes (extraction de $5.a_0\,d_3$)



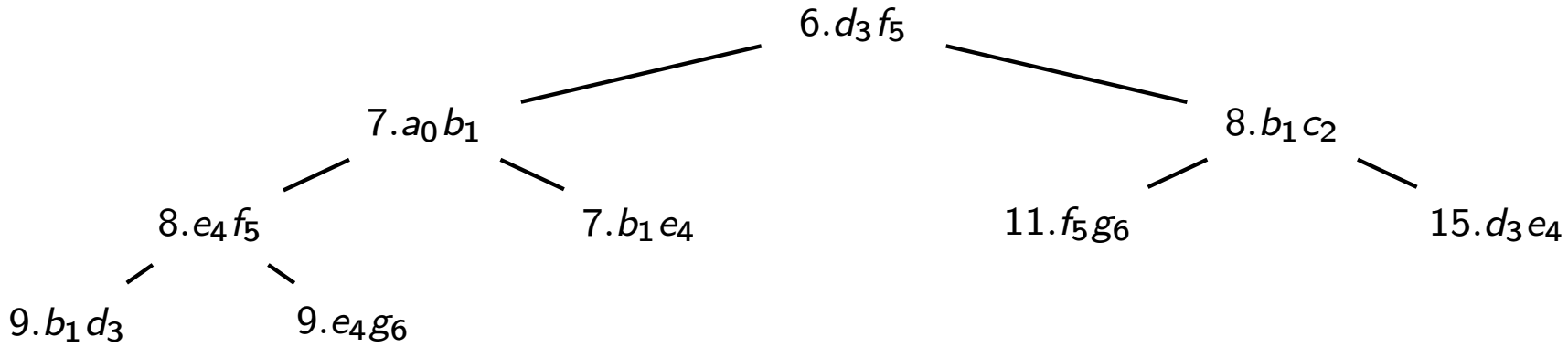
Déroulement de l'algorithme - boucle while



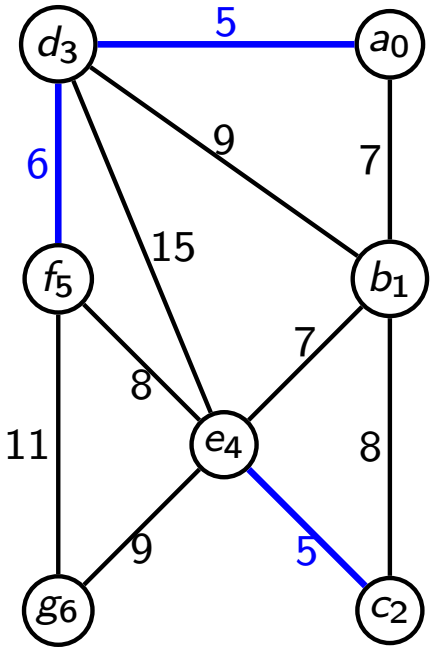
union-find des sommets



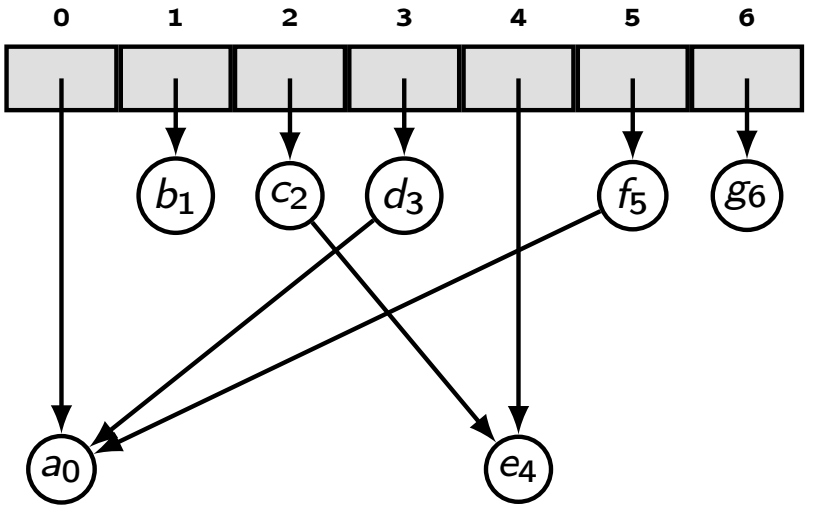
Tas min des arêtes (extraction de $5.c_2\,e_4$)



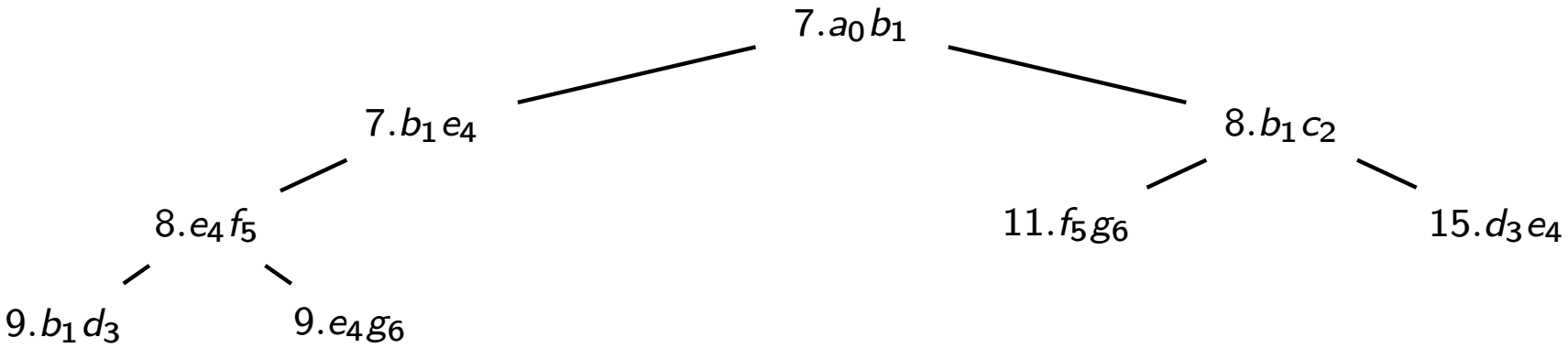
Déroulement de l'algorithme - boucle while



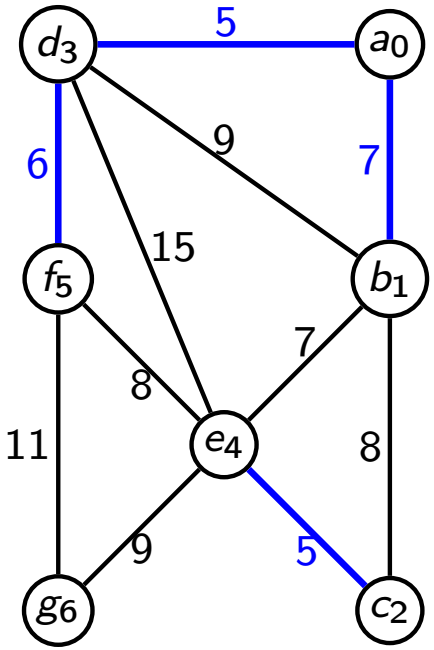
union-find des sommets



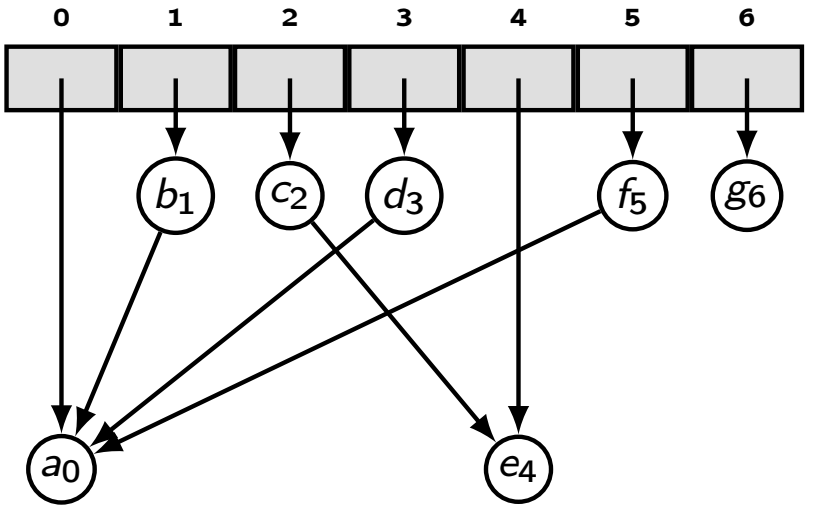
Tas min des arêtes (extraction de $6.d_3\,f_5$)



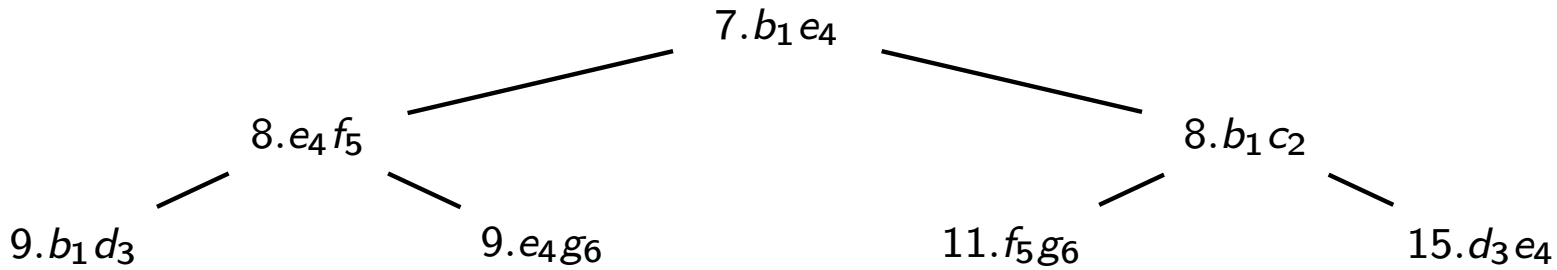
Déroulement de l'algorithme - boucle while



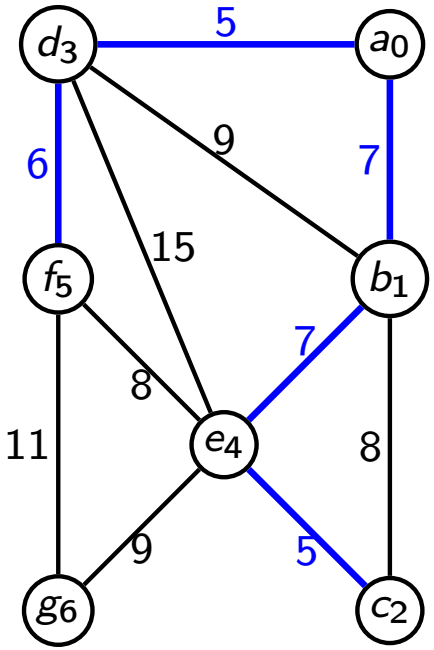
union-find des sommets



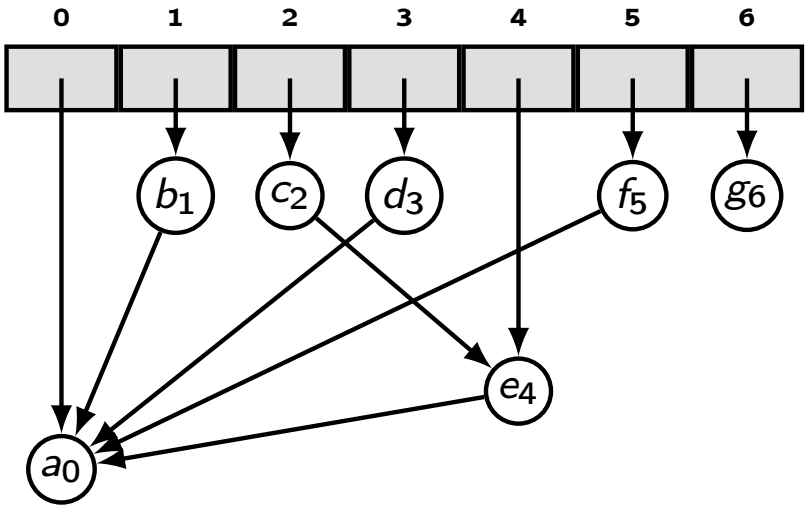
Tas min des arêtes (extraction de $7.a_0\,b_1$)



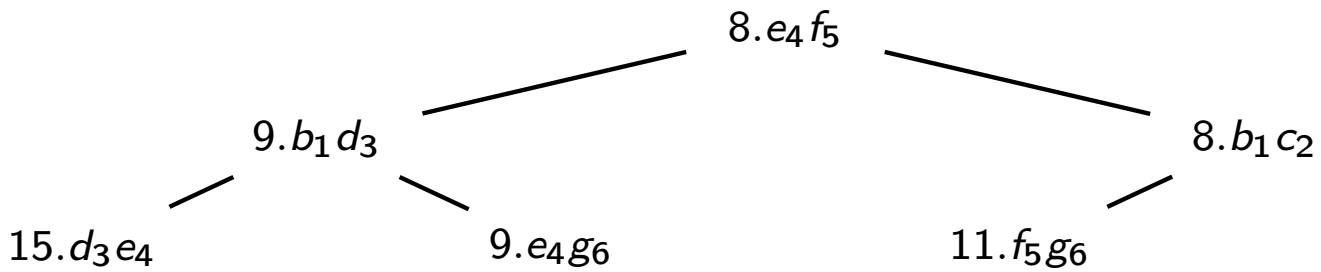
Déroulement de l'algorithme - boucle while



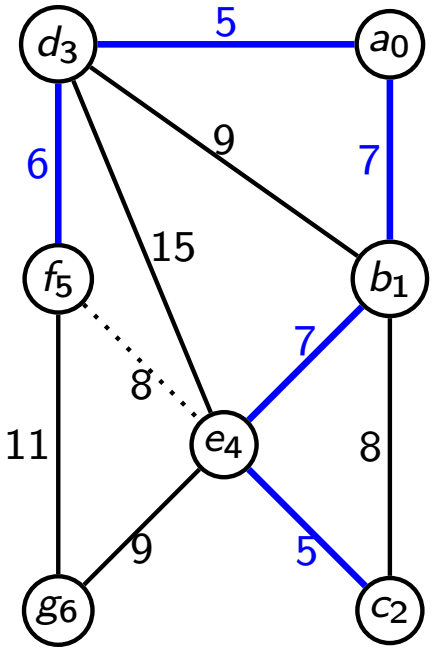
union-find des sommets



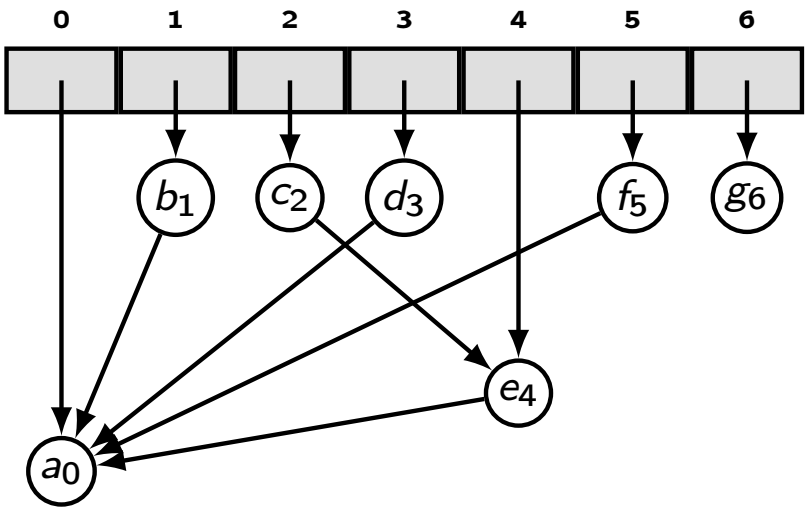
Tas min des arêtes (extraction de $7.b_1\,e_4$)



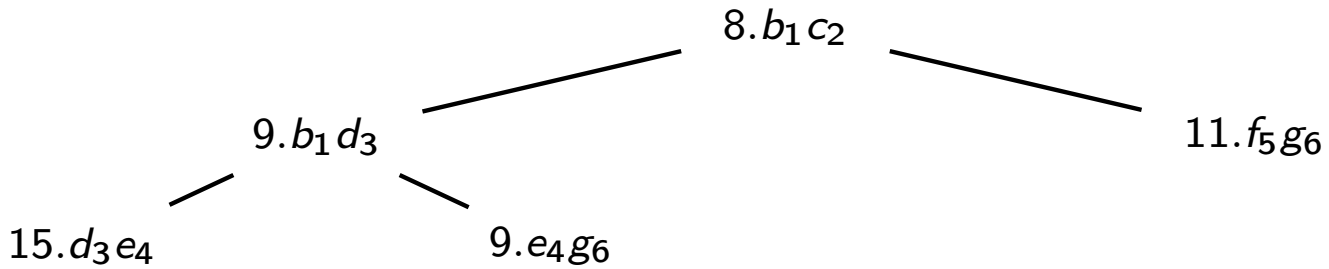
Déroulement de l'algorithme - boucle while



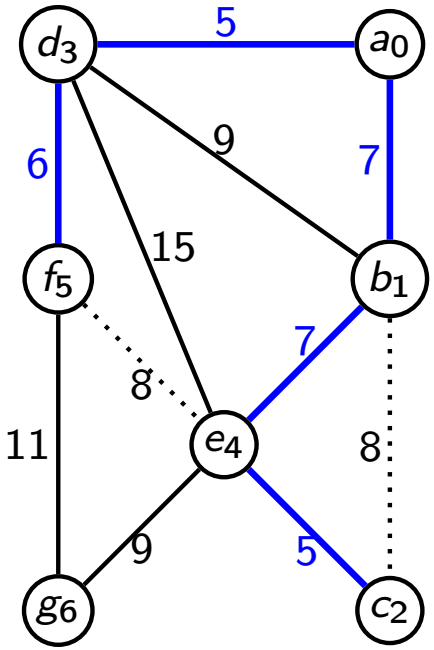
union-find des sommets



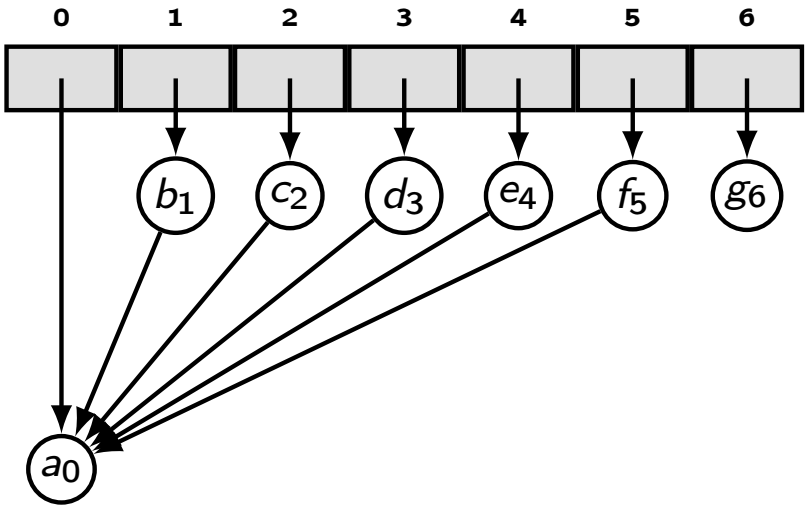
Tas min des arêtes (extraction de $8.e_4f_5$)



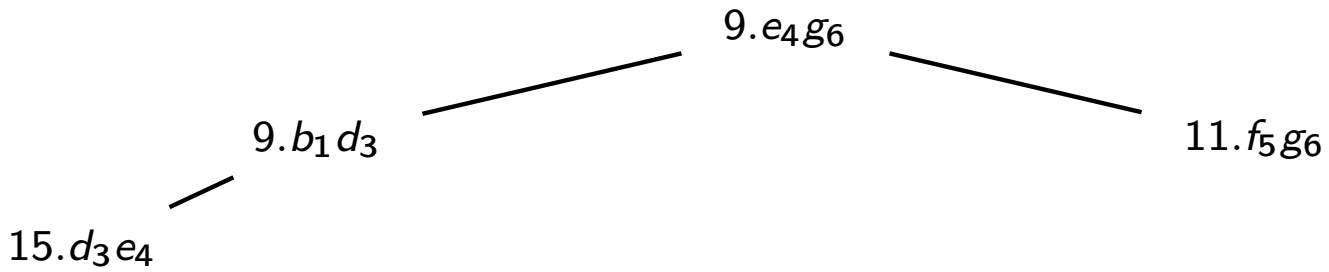
Déroulement de l'algorithme - boucle while



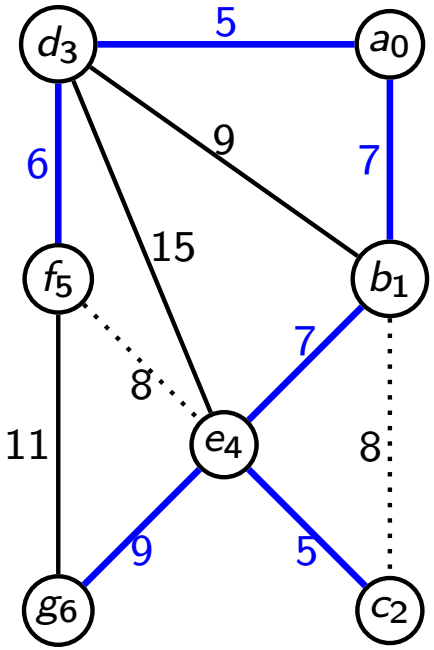
union-find des sommets



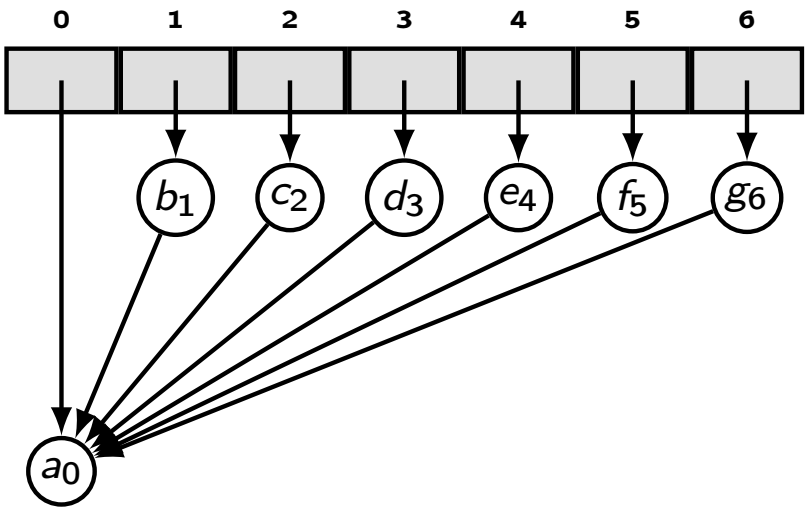
Tas min des arêtes (extraction de $8.b_1c_2$)



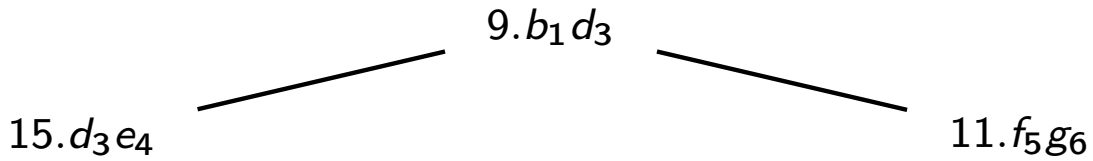
Déroulement de l'algorithme - boucle while



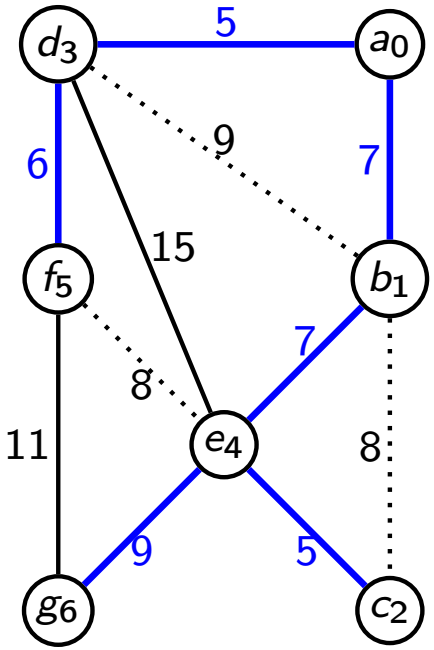
union-find des sommets



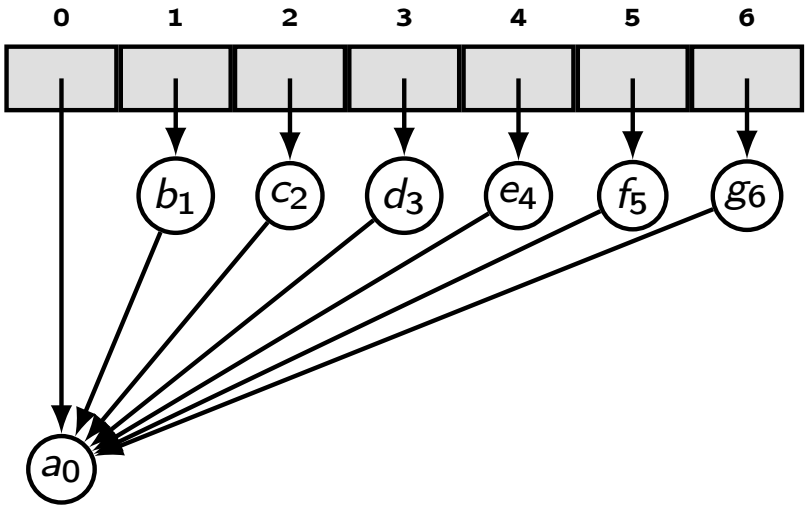
Tas min des arêtes (extraction de $9.e_4g_6$)



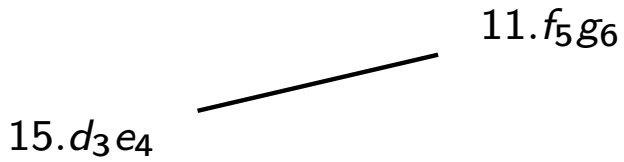
Déroulement de l'algorithme - boucle while



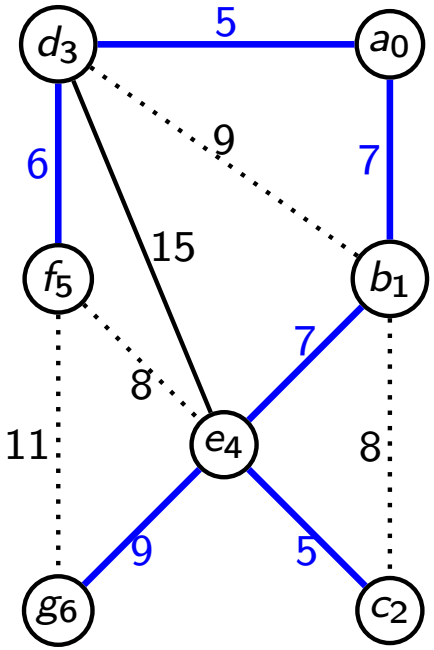
union-find des sommets



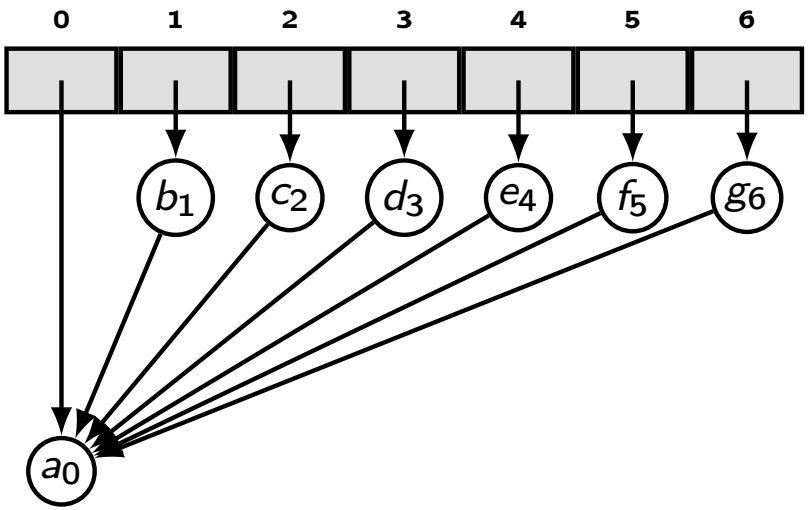
Tas min des arêtes (extraction de $9.b_1\,d_3$)



Déroulement de l'algorithme - boucle while



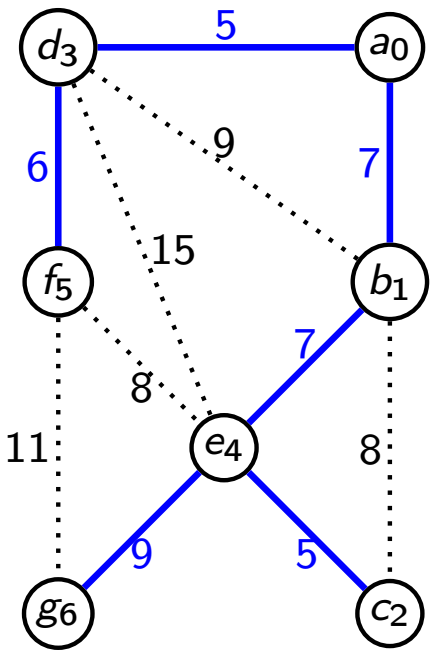
union-find des sommets



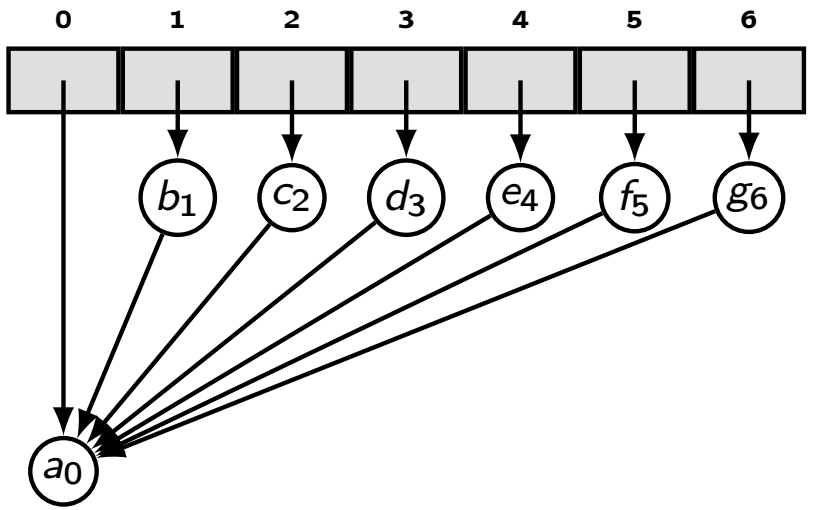
Tas min des arêtes (extraction de $11.f_5g_6$)

$15.d_3e_4$

Déroulement de l'algorithme - boucle while



union-find des sommets



Tas min des arêtes (extraction de $15.d_3e_4$ et fin)

Résultat final

- L'arbre couvrant est donné par la liste d'arêtes

$$(a_0d_3), (c_2e_4), (d_3f_5), (a_0b_1), (b_1e_4), (e_4g_6)$$

- C'est un arbre non enraciné, c'est-à-dire que la racine n'est pas définie

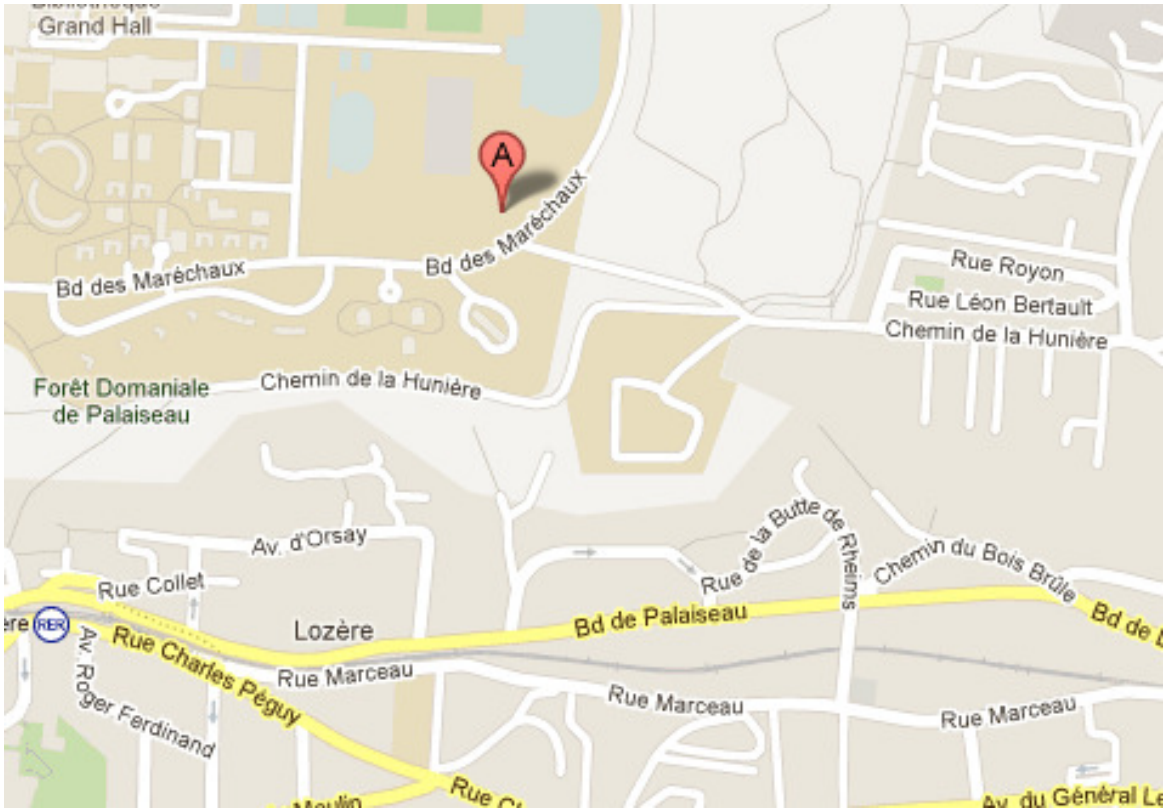
API des algorithmes sur les graphes d'entiers

API algorithmes de Kruskal

```
typedef struct we_ {
  int source;
  int destination;
  double weight;
} we_t;

int integer_mst(integer_graph_t *graph, generic_list_t **span);
```

Plus court chemin

Motivation : assistance au déplacement



Objectif : trouver un chemin entre un point A et un point B

- qui est le plus court en distance
- ou qui le plus court en temps
- ou qui est le plus court en ...

Différentes classes de problèmes

- Plus court chemin à origine unique et poids positifs ou nuls :

  algorithme de Dijkstra

  considéré dans ce cours
- Plus court chemin à origine unique et poids quelconques :

  algorithme de Bellman-Ford [2]
- Plus court chemin entre toutes les paires de sommets et poids quelconques (modulo détails) :

  algorithme de Floyd-Warshall [2]

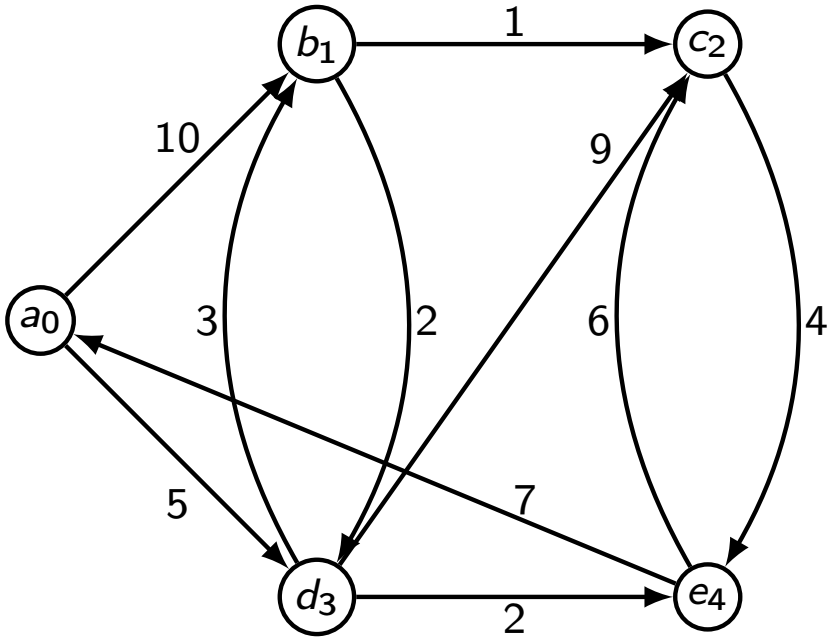
2. *cf* RO201 en 2A Math ou RO202 en 2A Info pour plus de détails

Algorithme de Dijkstra

Pseudo-code

```
Function shortest(graph G, start) :
    for /* Tous les sommets s de S */ do
        d[s]= ∞;
        π[s] = -1;
    end
    d[start] = 0;
    for /* Toutes les sommets s de S */ do
        heap_insert (&heap_min, (s, d[s],π[s]));
    end
    while heap_size (&heap_min) > 0 do
        u = heap_extract (&heap_min);
        list_ins_next (&list, list_tail(&list), u);
        for Tous les voisins v de u do
            if d[v] > d[u] + poids(u,v) then
                d[v] > d[u] + poids(u,v);
                π[v] = u;
            end
        end
    end
```

Déroulement de l'algorithme



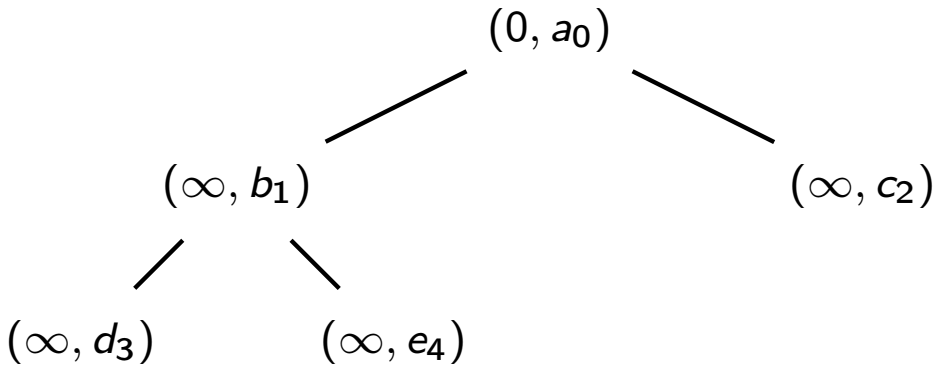
- Tableau des distances d

0	1	2	3	4
0	∞	∞	∞	∞

- Tableau des parents π

0	1	2	3	4
-1	-1	-1	-1	-1

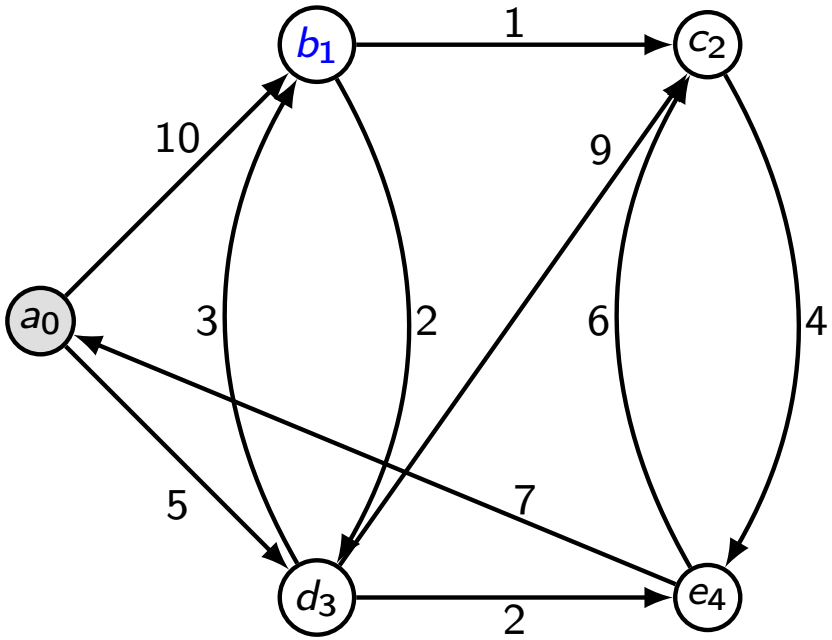
- Tas-min des (distances, sommets)



Étapes de l'algorithme

- Après les deux boucles for

Déroulement de l'algorithme



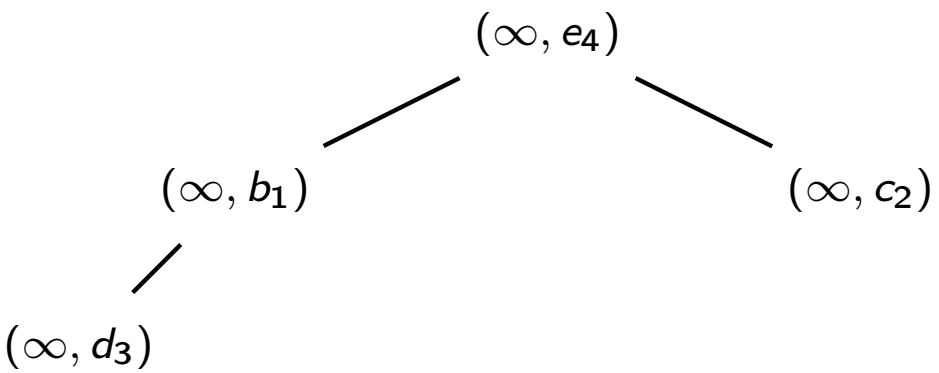
- Tableau des distances d

0	1	2	3	4
0	∞	∞	∞	∞

- Tableau des parents π

0	1	2	3	4
-1	-1	-1	-1	-1

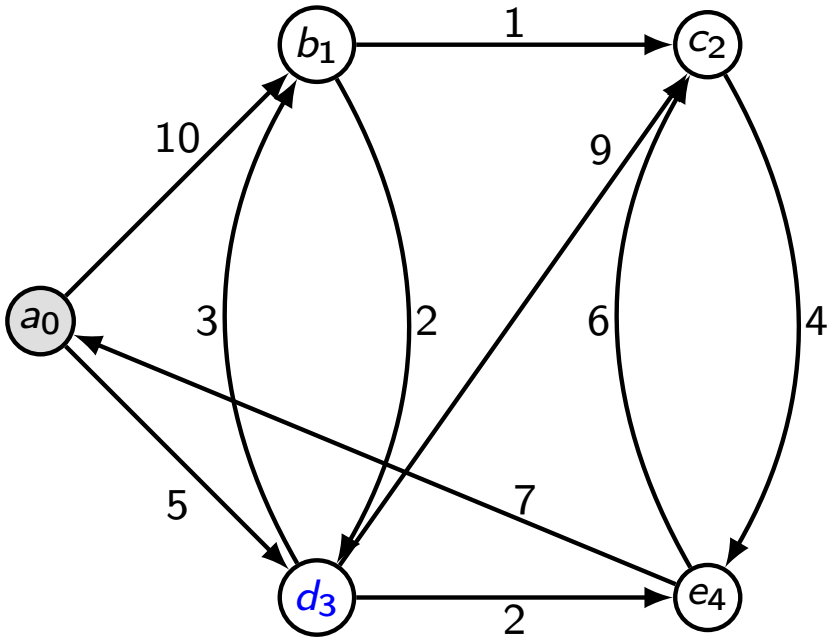
- Tas-min extraction de $(0, a_0)$



Étapes de l'algorithme

- Après les deux boucles for
- Parcours du graphe (while)
  - Examen voisins : b_1, d_3
  - Mise à jour des distances

Déroulement de l'algorithme



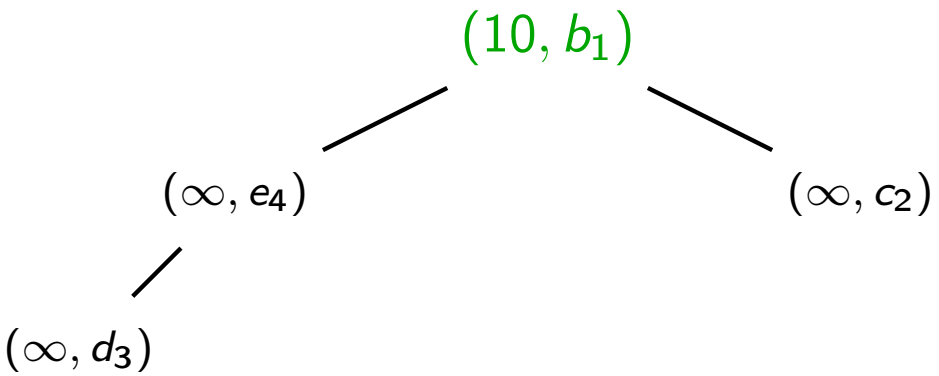
- Tableau des distances d

0	1	2	3	4
0	10	∞	∞	∞

- Tableau des parents π

0	1	2	3	4
-1	0	-1	-1	-1

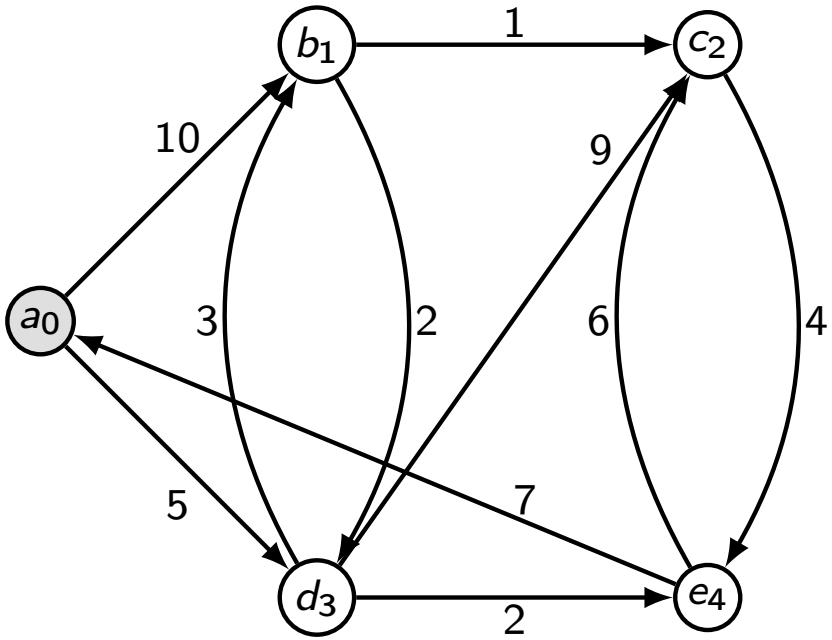
- Tas-min modification distance b_1



Étapes de l'algorithme

- Après les deux boucles for
- Parcours du graphe (while)
  - Examen voisins : b_1, d_3
  - Mise à jour des distances
    - $d[1] > d[0] + w(0,1)$?

Déroulement de l'algorithme



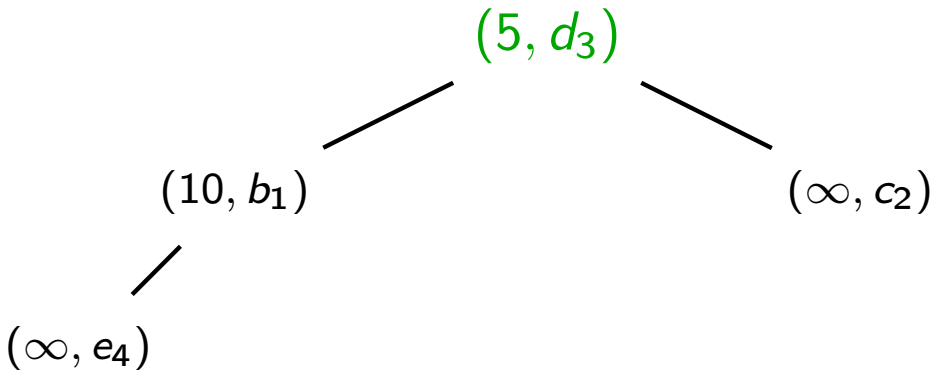
- Tableau des distances d

0	1	2	3	4
0	10	∞	5	∞

- Tableau des parents π

0	1	2	3	4
-1	0	-1	0	-1

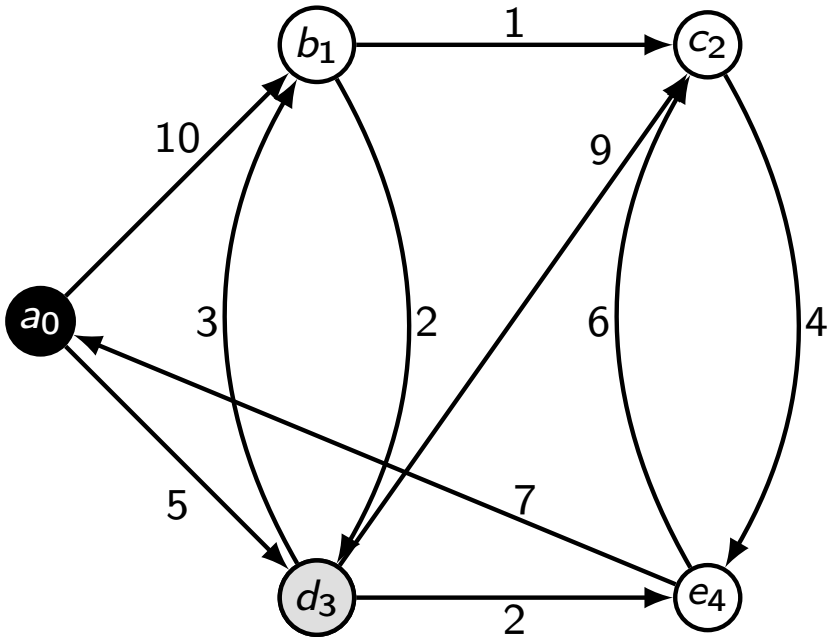
- Tas-min modification distance d_3



Étapes de l'algorithme

- Après les deux boucles for
- Parcours du graphe (while)
  - Examen voisins : b_1, d_3
  - Mise à jour des distances
    - $d[1] > d[0] + w(0,1)$?
    - $d[3] > d[0] + w(0,3)$?

Déroulement de l'algorithme



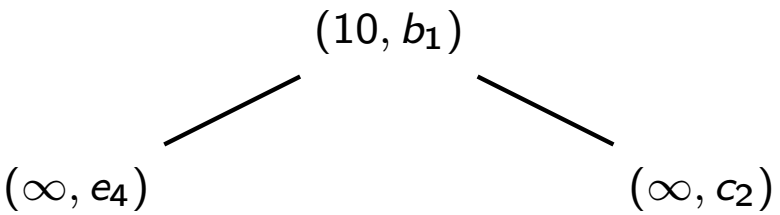
- Tableau des distances d

0	1	2	3	4
0	10	∞	5	∞

- Tableau des parents π

0	1	2	3	4
-1	0	-1	0	-1

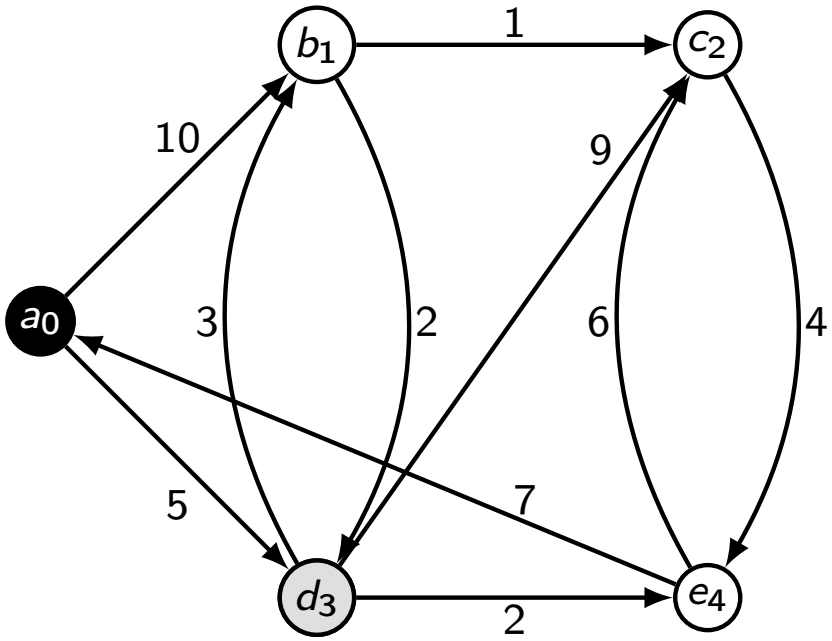
- Tas-min extraction de $(5, d_3)$



Étapes de l'algorithme

- Après les deux boucles for
- Parcours du graphe (while)
  - Examen voisins : b_1, c_2, e_4
  - Mise à jour des distances

Déroulement de l'algorithme



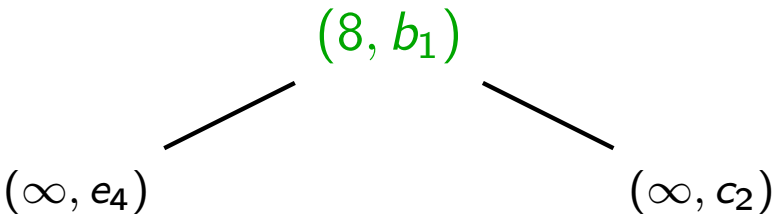
- Tableau des distances d

0	1	2	3	4
0	8	∞	5	∞

- Tableau des parents π

0	1	2	3	4
-1	3	-1	0	-1

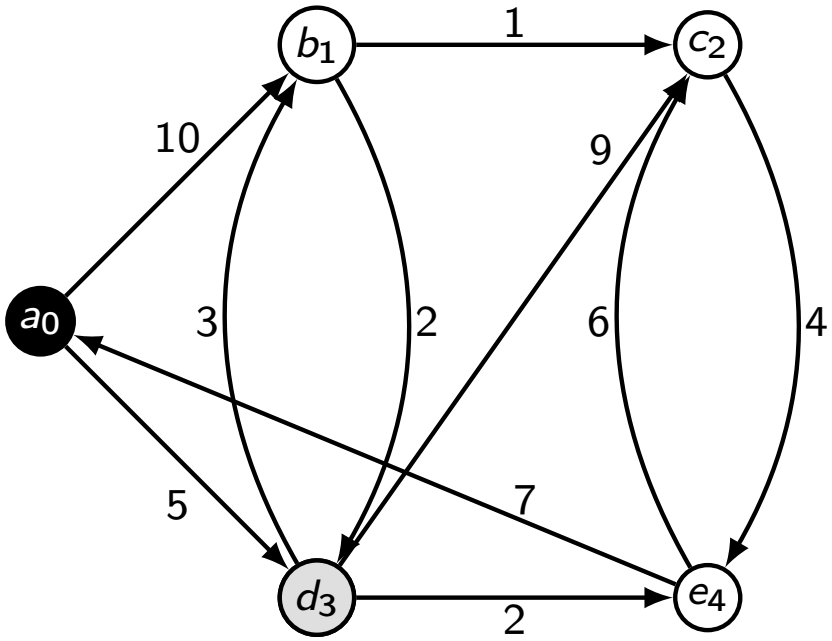
- Tas-min modification distance b_1



Étapes de l'algorithme

- Après les deux boucles for
- Parcours du graphe (while)
  - Examen voisins : b_1, c_2, e_4
  - Mise à jour des distances
    - $d[1] > d[3] + w(3,1)$?

Déroulement de l'algorithme



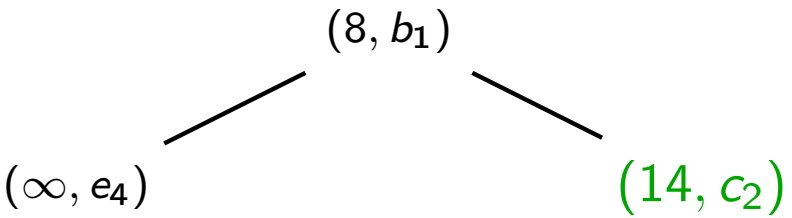
- Tableau des distances d

0	1	2	3	4
0	8	14	5	∞

- Tableau des parents π

0	1	2	3	4
-1	3	3	0	-1

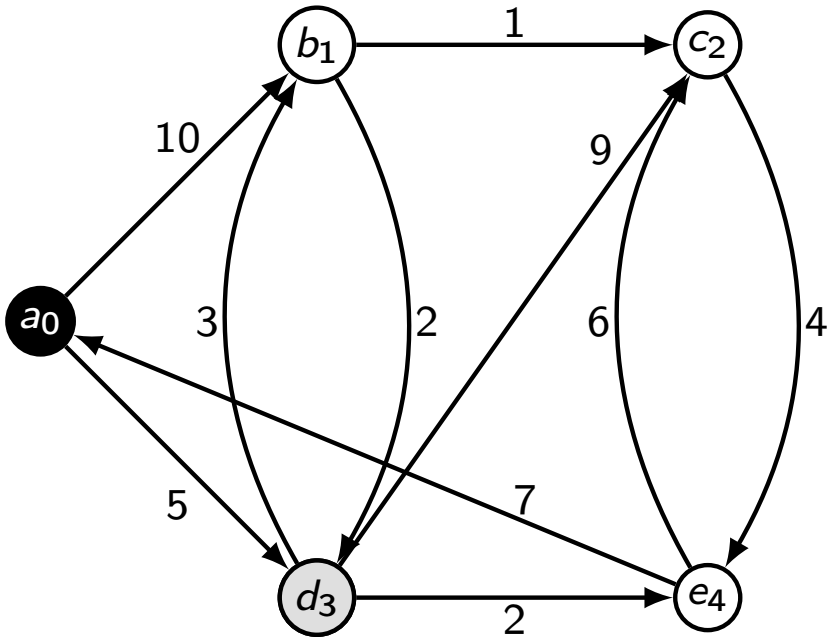
- Tas-min modification distance c_2



Étapes de l'algorithme

- Après les deux boucles for
- Parcours du graphe (while)
  - Examen voisins : b_1, c_2, e_4
  - Mise à jour des distances
    - $d[1] > d[3] + w(3,1)$?
    - $d[2] > d[3] + w(3,2)$?

Déroulement de l'algorithme



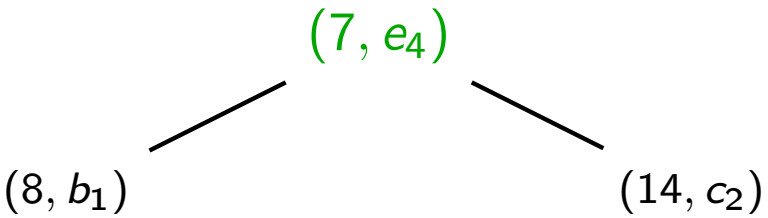
- Tableau des distances d

0	1	2	3	4
0	8	14	5	7

- Tableau des parents π

0	1	2	3	4
-1	3	3	0	3

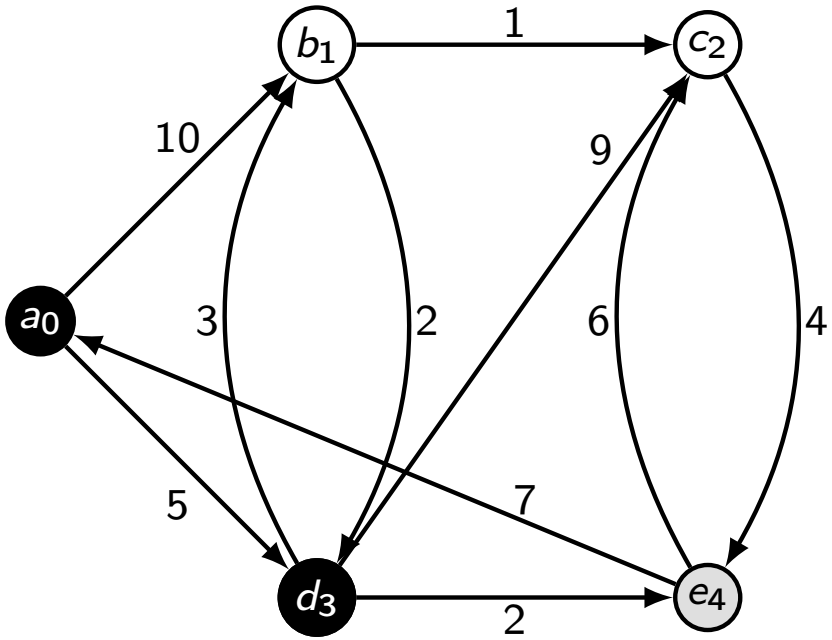
- Tas-min modification distance e_4



Étapes de l'algorithme

- Après les deux boucles for
- Parcours du graphe (while)
  - Examen voisins : b_1, c_2, e_4
  - Mise à jour des distances
    - $d[1] > d[3] + w(3,1)$?
    - $d[2] > d[3] + w(3,2)$?
    - $d[4] > d[3] + w(3,4)$?

Déroulement de l'algorithme



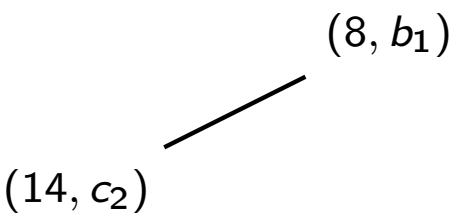
- Tableau des distances d

0	1	2	3	4
0	8	14	5	7

- Tableau des parents π

0	1	2	3	4
-1	3	3	0	3

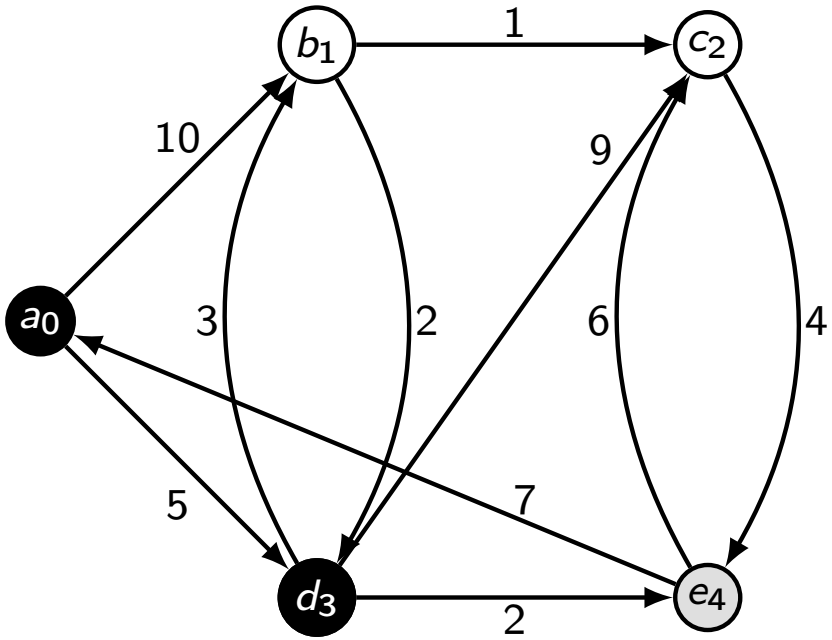
- Tas-min extraction de $(7, e_4)$



Étapes de l'algorithme

- Après les deux boucles for
- Parcours du graphe (while)
  - Examen voisins : a_0, c_2
  - Mise à jour des distances

Déroulement de l'algorithme



- Tableau des distances d

0	1	2	3	4
0	8	14	5	7

- Tableau des parents π

0	1	2	3	4
-1	3	3	0	3

- Tas-min pas de changement distance a_0

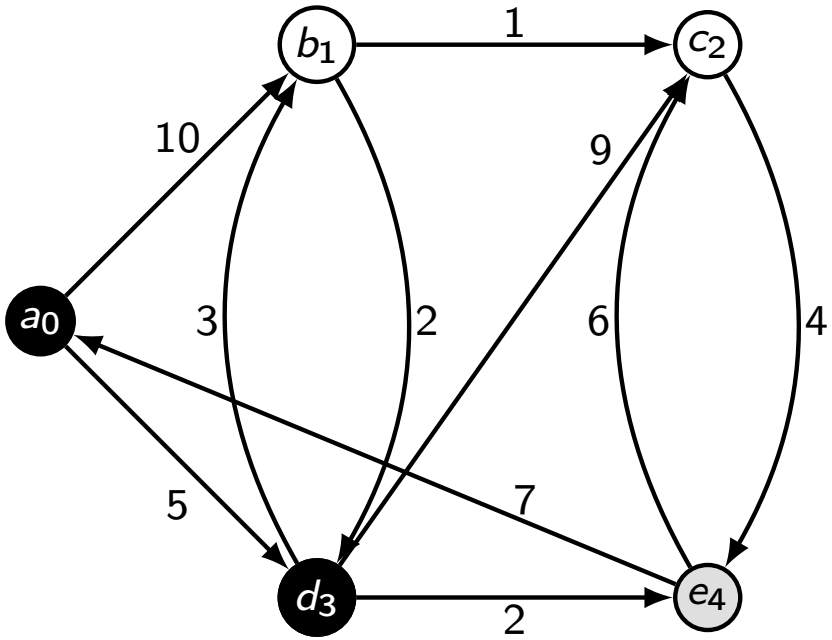
$(8, b_1)$

$(14, c_2)$

Étapes de l'algorithme

- Après les deux boucles for
- Parcours du graphe (while)
  - Examen voisins : a_0, c_2
  - Mise à jour des distances
    - $d[0] > d[4] + w(4, 0)$?

Déroulement de l'algorithme



- Tableau des distances d

0	1	2	3	4
0	8	13	5	7

- Tableau des parents π

0	1	2	3	4
-1	3	4	0	3

- Tas-min modification distance c_2

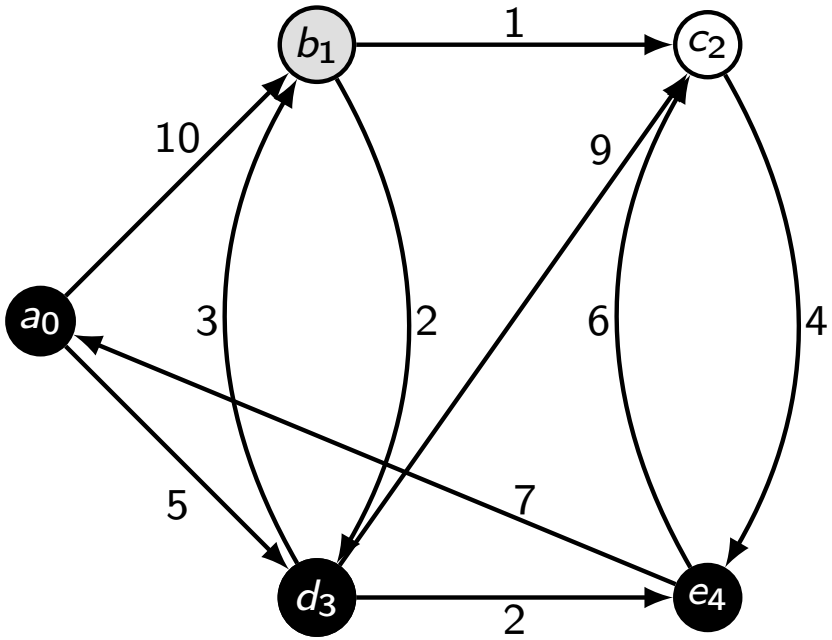
$(8, b_1)$

$(13, c_2)$

Étapes de l'algorithme

- Après les deux boucles for
- Parcours du graphe (while)
  - Examen voisins : a_0, c_2
  - Mise à jour des distances
    - $d[0] > d[4] + w(4, 0)$?
    - $d[2] > d[4] + w(4, 2)$?

Déroulement de l'algorithme



- Tableau des distances d

0	1	2	3	4
0	8	13	5	7

- Tableau des parents π

0	1	2	3	4
-1	3	4	0	3

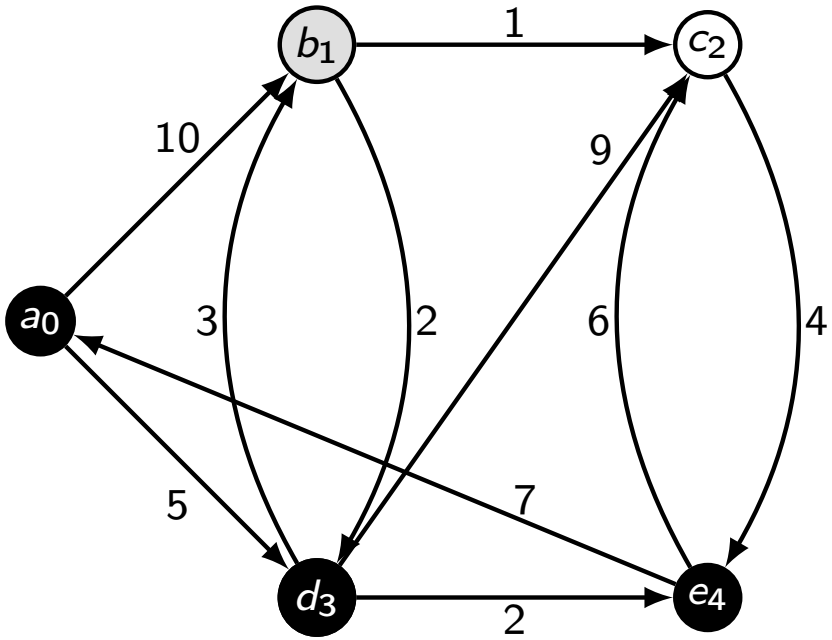
- Tas-min extraction de $(8, b_1)$

$(13, c_2)$

Étapes de l'algorithme

- Après les deux boucles for
- Parcours du graphe (while)
  - Examen voisins : c_2, d_3
  - Mise à jour des distances

Déroulement de l'algorithme



- Tableau des distances d

0	1	2	3	4
0	8	9	5	7

- Tableau des parents π

0	1	2	3	4
-1	3	1	0	3

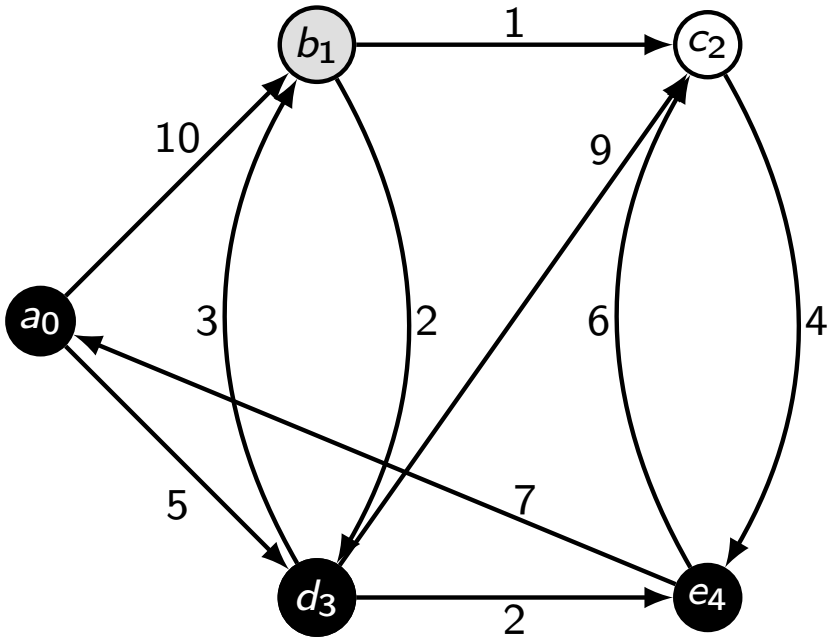
- Tas-min modification distance c_2

$(9, c_2)$

Étapes de l'algorithme

- Après les deux boucles for
- Parcours du graphe (while)
  - Examen voisins : c_2, d_3
  - Mise à jour des distances
    - $d[2] > d[1] + w(1,2)$?

Déroulement de l'algorithme



- Tableau des distances d

0	1	2	3	4
0	8	9	5	7

- Tableau des parents π

0	1	2	3	4
-1	3	1	0	3

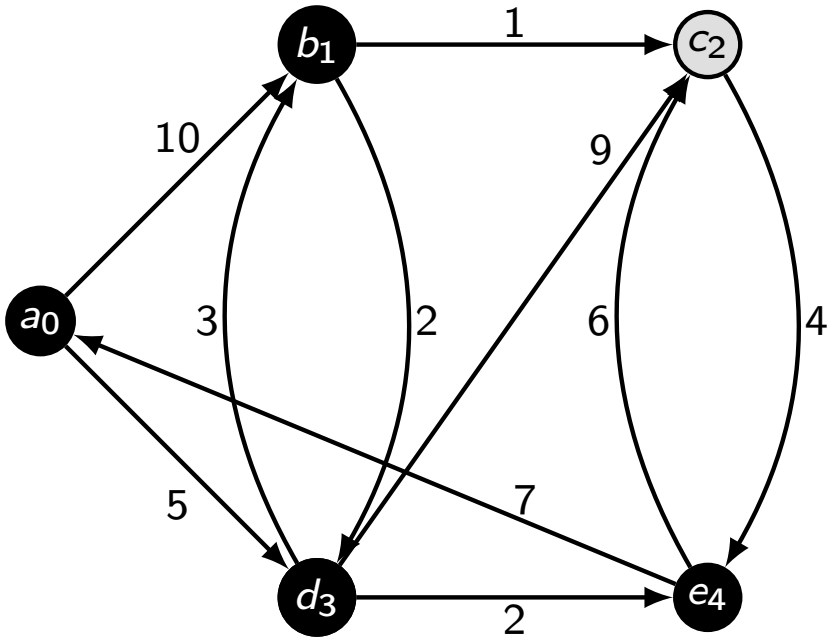
- Tas-min pas de changement distance d_3

$(9, c_2)$

Étapes de l'algorithme

- Après les deux boucles for
- Parcours du graphe (while)
  - Examen voisins : c_2, d_3
  - Mise à jour des distances
    - $d[2] > d[1] + w(1,2)$?
    - $d[3] > d[1] + w(1,3)$?

Déroulement de l'algorithme



- Tas-min extraction de $(9, c_2)$

- Tableau des distances d

0	1	2	3	4
0	8	9	5	7

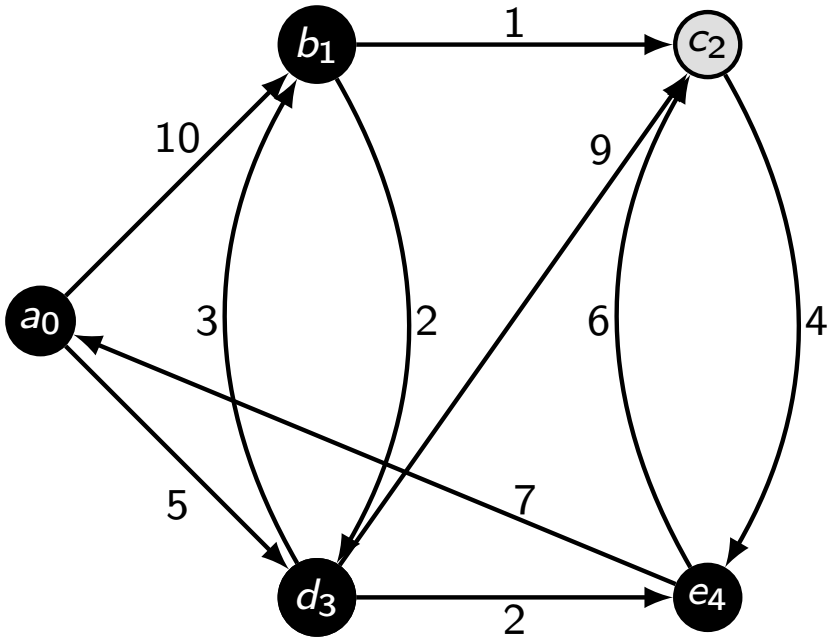
- Tableau des parents π

0	1	2	3	4
-1	3	1	0	3

Étapes de l'algorithme

- Après les deux boucles for
- Parcours du graphe (while)
  - Examen voisin : e_4
  - Mise à jour des distances

Déroulement de l'algorithme



- Tableau des distances d

0	1	2	3	4
0	8	9	5	7

- Tableau des parents π

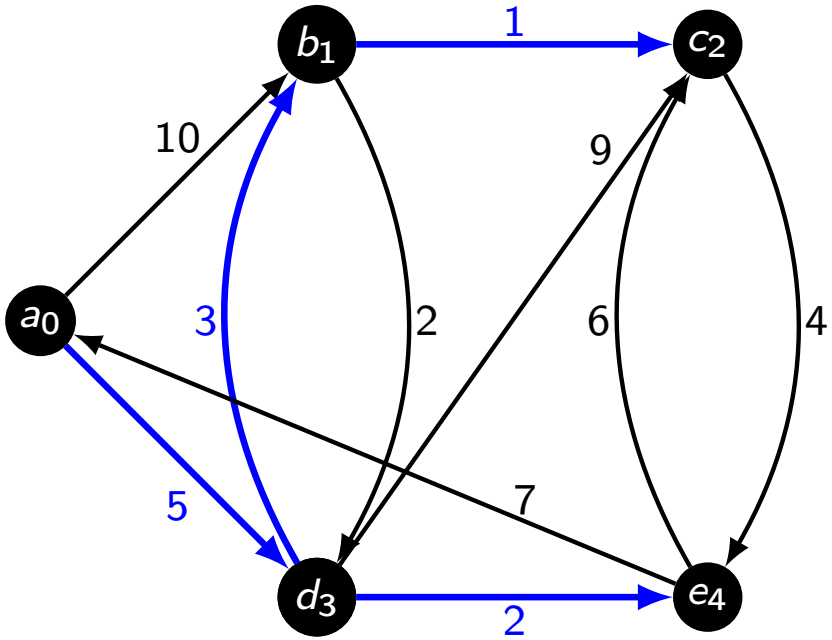
0	1	2	3	4
-1	3	1	0	3

- Tas-min pas de changement distance e_4

Étapes de l'algorithme

- Après les deux boucles for
- Parcours du graphe (while)
  - Examen voisin : e_4
  - Mise à jour des distances
    - $d[4] > d[2] + w(2,4)$?

Déroulement de l'algorithme



- Tableau des distances d

0	1	2	3	4
0	8	9	5	7

- Tableau des parents π

0	1	2	3	4
-1	3	1	0	3

Au final, on a calculé tous les chemins depuis a_0 vers les autres sommets de coût minimal, représentés en bleu (pour simplicité), mais donnés par le tableau π.

Étapes de l'algorithme

- Après les deux boucles for
- Parcours du graphe (while)
- Fin

API des algorithmes sur les graphes d'entiers

API algorithmes Dijkstra

```
typedef struct ed_ {
  int vertex;
  double distance;
  int parent;
} ed_t;

int integer_shortest(integer_graph_t *graph, int start,
                     generic_list_t **paths);
```

MST vs Plus court chemin

	MST	Plus court chemin
Définition	Un arbre qui traverse tous les sommets, tout en minimisant le poids total.	Le chemin dont la distance accumulées est la plus faible.
Objectif	Connexion de tous les nœuds entre eux avec un poids total minimal.	Trouver l'itinéraire le plus efficace entre deux sommets.
Applications	La conception de réseaux pour la pose de câbles (minimise le coût)	La connexion entre les deux points peut être choisie très efficacement par le chemin le plus court et cela aide également les gens à naviguer vers l'endroit final avec une distance minimale.

Conclusion

En IN103, nous avons vu beaucoup beaucoup de choses

- des structures de données :
  - listes, piles, files, ensembles, arbres, tas, union-find, graphe
- des algorithmes pour résoudre des problèmes classiques :
  - les itérateurs, les parcours d'arbres ou de graphes, couverture d'ensemble, plus court chemin, arbre couvrant de poids minimal, les composantes (fortement) connexes, coloration de graphe

Remarque ;-)

Tout ceci n'est (encore) qu'un petit aperçu des possibilités infinies de la science informatique.