

PROGRAMMATION AVANCÉE

PHUC NGO

CONTENUE DU COURS

- Rappels
 - Fonctions et procédures
 - Structures de données
 - Enregistrements, tableaux, liste enchainé, Pile(FIFO), file(FILO), ...
- Compilation avec Makefile/Cmake et la bibliothèque SDL
- Gestion de mémoire
 - Représentation de mémoire
 - Pointeurs et allocation dynamique
- Gestion de fichiers
 - Lecture et écriture
- Gestion des entrées et sorties
 - Ecran, souris et clavier
- **Directives au préprocesseur**
- Structures de données et algorithmes de graphes
 - Parcours de graphe, arbre couvrant, ...

DIRECTIVES AU PRÉPROCESSEUR

- Le préprocesseur est un programme exécuté lors de la première phase de la compilation
- Les directives au préprocesseur, introduites par **#**
 - L'incorporation de fichiers source : **#include**
 - La définition de constantes symboliques et de macros : **#define**
 - La compilation conditionnelle : **#if**, **#ifdef**, **#ifndef**, **#endif**, ...
- Les directives effectuent des modifications textuelles sur le fichier source

DIRECTIVES AU PRÉPROCESSEUR

- La directive **#include**
 - Incorporer dans le fichier source d'autre fichier
 - Fichiers en-têtes de la bibliothèque standard : stdio.h, math.h, ...
 - Fichiers créés par l'utilisateur
 - Syntaxes :
 - Recherche dans un ou plusieurs répertoires systèmes (ex : usr/include) : **#include <nom-de-fichier>**
 - Exemple : #include <stdio.h>
 - Recherche dans le répertoire courant ou le répertoire définit avec l'option -I du compilateur : **#include "nom-de-fichier"**
 - Exemple : #include "mes_fonctions.h"

DIRECTIVES AU PRÉPROCESSEUR

- La directive **#define**
 - Définir des constantes symboliques
 - Définir des macros avec paramètres
- Syntaxe
 - Constantes symbolique : **#define nom valeur**
 - Exemple : #define TAILLE_MAX 20

```
#define BEGIN {  
#define END }
```
 - Macros **#define nom(paramètres) (corps-de-la-macro)**
 - Exemple : #define MAX(a,b) (a > b ? a : b) → MAX(5,7) = 7

```
#define CARRE (a) (a * a) → CARRE(2) = 2*2 = 4
```
 - Attention : CARRE(x+y) ⇔ (x+y*x+y) !!! MAIS PAS (x+y)*(x+y)
 - Définition correcte : #define CARRE (a) ((a)*(a))

DIRECTIVES AU PRÉPROCESSEUR

- La compilation conditionnelle **#ifdef**
 - Exécution du programme en fonction de directives
- Syntaxe :

```
#if condition-1  
    partie-du-programme-1  
#elif condition-2  
    partie-du-programme-2  
...  
#elif condition-n  
    partie-du-programme-n  
#else  
    partie-par-défault  
#endif
```

Exemple :

```
#ifdef DEBUG  
    for (i = 0; i < N; i++)  
        printf("%d\n",i);  
#endif /* DEBUG */
```

Dans le programme :
#define DEBUG

A la compilation :
gcc -DDEBUG fichier.c

DIRECTIVES AU PRÉPROCESSEUR

- La compilation conditionnelle **#ifndef**
 - Le préprocesseur détermine si la macro fournie n'existe pas avant d'inclure le code dans le processus de compilation
 - Syntaxe : **#ifndef def_macro**
 1. Les fichiers d'entêtes

```
#ifndef MON_HEADER_H  
#define MON_HEADER_H  
// Contenu du mon_header.h  
#endif //MON_HEADER_H
```
 2. Les jetons

```
#ifndef JETON  
// Code si le jeton n'est pas défini  
#else  
// Code si le jeton est défini  
#endif
```

CONTENUE DU COURS

- Rappels
 - Fonctions et procédures
 - Structures de données
 - Enregistrements, tableaux, liste enchainé, Pile(FIFO), file(FILO), ...
- Compilation avec Makefile et la bibliothèque SDL
- Gestion de mémoire
 - Représentation de mémoire
 - Pointeurs et allocation dynamique
- Gestion de fichiers
 - Lecture et écriture
- Gestion des entrées et sorties
 - Ecran, souris et clavier
- Directives au préprocesseur
- **Structures de données** et algorithmes de graphes
 - Parcours de graphe, arbre couvrant, ...

STRUCTURE

- Structures = enregistrement
 - Structure permettant de **combiner** des éléments de données de différents types
 - Une instance d'une enregistrement prend une mémoire au moins la **somme** de la taille des éléments de l'enregistrement
 - Syntaxe :

```
struct [nom_structure]{
    type champ_1;
    ...
    type champ_n;
};
```

```
struct nom_structure variable;
variable.champ_1 = valeur_1;
valeur_2 = variable.champ2;
```

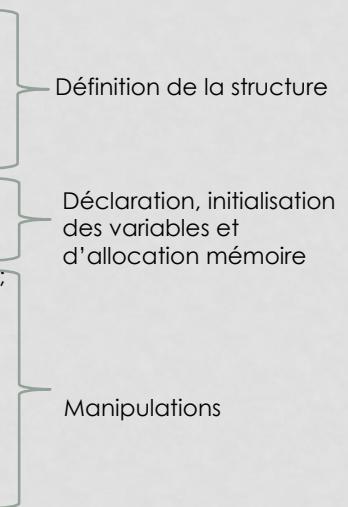
```
typedef struct {
    type champ_1;
    ...
    type champ_n;
} [nom_structure];
```

```
nom_structure variable;
variable.champ_1 = valeur_1;
valeur_2 = variable.champ2;
```

EXEMPLE : STRUCTURES

```
typedef struct {
    char nom[20];
    char prenom[20];
    bool sexe;
    int age;
} personne;
personne p = { "Paul", "Jean", True, 20};
personne p1;
personne* p2=malloc(sizeof(personne));
printf("%s - %s : %d", p.nom, p.prenom, p.age);
p1.age=19;
(*p2).age=19; ←→ p2→age=19
printf("%d et %d", p1.age, p2→age);

printf( "nom p1: " );
scanf( "%s", &p1.nom );
printf( "nom p2: " );
scanf( "%s", &(*p2). nom ); ←→ &(p2→nom)
```



EXEMPLE : STRUCTURES

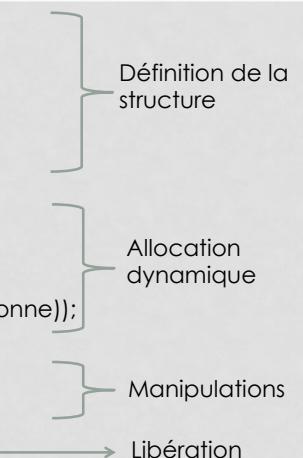
```
typedef struct {
    char nom[20];
    char prenom[20];
    bool sexe;
    int age;
} personne;
personne p = { "Paul", "Jean", True, 20};
personne p1;
personne* p2=malloc(sizeof(personne));
printf("%s - %s : %d", p.nom, p.prenom, p.age); → Paul - Jean : 20
p1.age=19;
(*p2).age=20; ←→ p2→age=20
printf("%d et %d", p1.age, p2→age); → 19 et 20

printf( "nom p1: " );
scanf( "%s", &p1.nom );
printf( "nom p2: " );
scanf( "%s", &(*p2). nom ); ←→ &(p2→nom)
```

EXEMPLE : TABLEAU DE STRUCTURES

```
typedef struct {
    char nom[20];
    char prenom[20];
    bool sexe;
    int age;
} personne;
int nbPers;
printf( "saisir nombre de personne : " );
scanf( "%d", &nbPers );
personne* tabPers=malloc(nbPers*sizeof(personne));

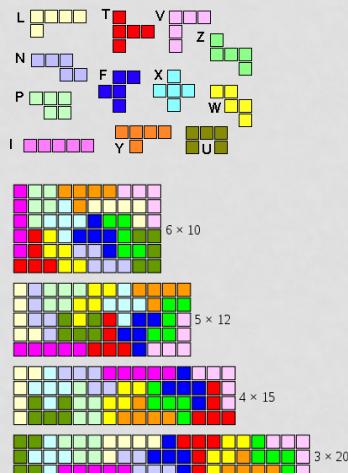
for (i=0; i<nbPers; i++)
    //saisir tabPers[i].nom,...
```



STRUCTURES POUR LE PROJET

- Structure pour une pièce
 - Nom : chaîne des caractères
 - Largeur et hauteur : int
 - Orientation : int
 - Position dans la grille : bool
 - Bon position dans la grille : bool
 - Coordonnées des carrées: tableau des coordonnées x et y

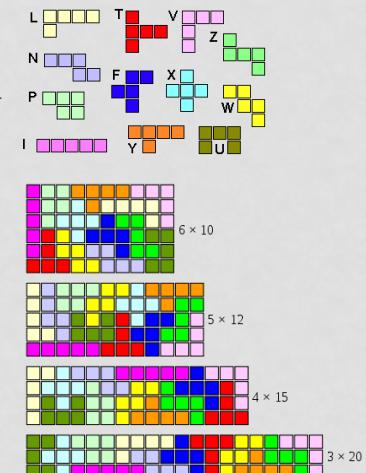
```
typedef struct {
    char nom[20];
    int l,h,orientation;
    int pos_x,pos_y;
    bool dans_grille, bon_pos;
    int** coord;
} piece;
```



STRUCTURES POUR LE PROJET

- Structure pour la grille
 - Nom : chaîne des caractères
 - Nombre de lignes et colonnes : int
 - Coordonnées x et y : int
 - Remplie : bool

```
typedef struct {
    char nom[20];
    int l,h;
    int pos_x,pos_y;
    bool est_rempli;
} grille;
```

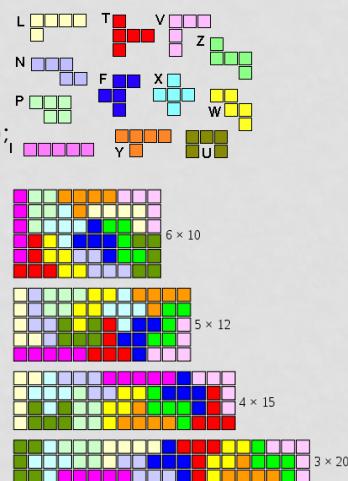


ALLOCATION DYNAMIQUE POUR LE PROJET

- Allocation dynamique pour la structure des pièces


```
piece* p1=malloc(sizeof(piece));
piece* p2=malloc(nbPiece*sizeof(piece));
```
- Allocation dynamique pour la structure de la grille pièce


```
grille* g=malloc(sizeof(gille));
```
- Les structures de données utilisées dans le projet
 - Liste chainée, pile, file, graphe, ...



UNIONS

- Unions
 - Structure permettant de stocker différents types de données dans le **même emplacement** de mémoire
 - Une union peut contenir plusieurs éléments, mais un seul élément peut contenir **une valeur à un moment donné**
 - Une instance d'une union prend une mémoire égale à la taille de l'élément le plus important de l'union
 - Syntaxe :

```
union [nom_union]{
    type champ_1;
    ...
    type champ_n;
}
union nom_union variable;
variable.champ_1 = valeur_1;
valeur_2 = variable.champ_2;
```

```
typedef union {
    type champ_1;
    ...
    type champ_n;
} [nom_union];
nom_union variable;
variable.champ_1 = valeur_1;
valeur_2 = variable.champ_2;
```

EXEMPLE : UNIONS

```

typedef union {
    int entier;
    float reel;
} nombre;
nombre n = {12}; //initialisation l'union
nombre* p= &n; // l'adresse de l'union

n.entier = 19;
printf( "valeur entiere = %d ", n.entier);

(*p).reel = 1200.5; ↔ p→reel=1200.5
printf( "valeur reelle = %f ", p→reel);
printf( "valeur reelle = %f ", n.reel);
printf( "valeur entiere = %d ", n.entier);

scanf( " n-entier %d", & n.entier);
scanf( "p-reel %f", &(*p). reel ); ↔ &(p→reel)

```

Définition de l'union
Déclaration des variables et d'allocation mémoire
Manipulations

EXEMPLE : UNIONS

```

typedef union {
    int entier;
    float reel;
} nombre;
nombre n = {12}; //initialisation l'union
nombre* p= &n; // l'adresse de l'union

n.entier = 19;
printf( "valeur entiere = %d ", n.entier); → valeur entiere = 19

(*p).reel = 1200.5; ↔ p→reel=1200.5
printf( "valeur reelle = %f ", p→reel); → valeur reelle = 1200.5
printf( "valeur reelle = %f ", n.reel); → valeur reelle = 1200.5
printf( "valeur entiere = %d ", n.entier); → valeur entiere = 1150681498

scanf( " n-entier %d", & n.entier);
scanf( "p-reel %f", &(*p). reel ); ↔ &(p→reel)

```

STRUCTURE ET UNION

	Struct	Union
Mémoire	Individuel = Une variable pour chaque champs	Partagé = même zone mémoire pour tous les champs
Taille	Somme totale des champs	Champs de taille max
Adresse	Adresse des champs est en ordre croissant	Adresse est la même pour tous les champs
Accès	Chaque champ peut être consulté à la fois	Un seul champs est accessible à la fois
Initialisation	Les champs d'une structure peuvent s'initialiser dans l'ordre	Seul le premier champ peut être initialisé
Modif.	Indépendant entre les champs	Altérant la valeur d'un champs modifiera d'autres champs

STRUCTURE AUTO-RÉFÉRENCEÉE

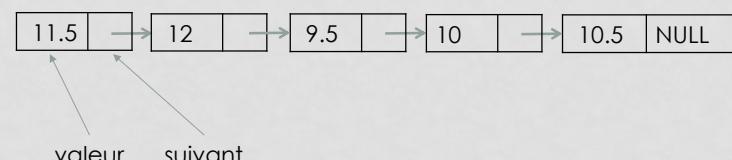
- Structure contient un pointeur vers la structure de même modèle

```

struct cellule
{
    int valeur;
    struct cellule *suivant;
};
typedef struct cellule *liste;

```

- Exemple : une liste



LISTE CHAINÉE

```
liste = tête
[11.5] → [12] → [9.5] → [10] → [10.5] → NULL
queue

typedef struct Element* liste;
typedef struct Element
{
    int premier;
    liste suivant;
} Element;
liste L=NULL;

bool est_vide (liste L)
{
    return L == NULL;
}
```

LISTE CHAINÉE

```
liste = tête
[11.5] → [12] → [9.5] → [10] → [10.5] → NULL
queue

typedef struct Element* liste;
typedef struct Element
{
    int premier;
    liste suivant;
} Element;
liste L=NULL;

Element* insertion (int x, liste L)
{
    Element* e ;
    e = malloc (sizeof (*e)); // Réservation de mémoire
    e→premier = x;
    e→suivant = L;
    return e;
}
```

LISTE CHAINÉE

```
liste = tête
[11.5] → [12] → [9.5] → [10] → [10.5] → NULL
queue

typedef struct Element* liste;
typedef struct Element
{
    int premier;
    liste suivant;
} Element;
liste L=NULL;

int premier (liste L)
{
    if (!est_vide (L)) {
        printf ("La liste est vide !\n");
        exit (EXIT_FAILURE);
    }
    return L→premier;
}
```

LISTE CHAINÉE

```
liste = tête
[11.5] → [12] → [9.5] → [10] → [10.5] → NULL
queue

typedef struct Element* liste;
typedef struct Element
{
    int premier;
    liste suivant;
} Element;
liste L=NULL;

liste reste (liste L)
{
    return L→suivant;
}
```

LISTE CHAINÉE

```
liste = tête
[ 11.5 ] → [ 12 ] → [ 9.5 ] → [ 10 ] → [ 10.5 ] → NULL
queue

typedef struct Element* liste;
typedef struct Element
{
    int premier;
    liste suivant;
} Element;
liste L=NULL;

bool liberer_liste (liste L)
{
    if (est_vide (L))
        return false;
    liberer_liste (reste (L));
    free(L);
    return true;
}
```

LISTE CHAINÉE

```
liste = tête
[ 11.5 ] → [ 12 ] → [ 9.5 ] → [ 10 ] → [ 10.5 ] → NULL
queue

typedef struct Element* liste;
typedef struct Element
{
    int premier;
    liste suivant;
} Element;
liste L=NULL;

//Autres opérations:
//calculer la longueur de la liste
//rechercher un élément
//insérer à la fin de la liste
//vérifier/compter le nombre d'élément dans la liste
//...
```

LISTE CHAINÉE

```
liste = tête
[ 11.5 ] → [ 12 ] → [ 9.5 ] → [ 10 ] → [ 10.5 ] → NULL
queue

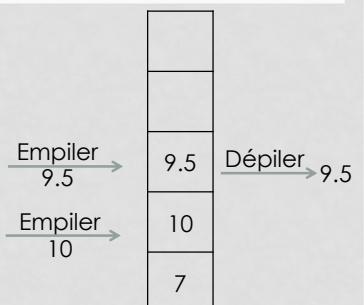
typedef struct Element* liste;
typedef struct Element
{
    int premier;
    liste suivant;
} Element;
liste L=NULL;

int main(void)
{
    if(est_vide(L))
        insertion(5,L);
    printf( " 1er element de la liste est %d ", premier(L) );
    liberer_liste (L);
    return EXIT_SUCCESS;
}
```

PILE (LIFO : LAST IN FIRST OUT)

```
typedef struct Element* pile;
typedef struct Element
{
    int premier;
    pile suivant;
} Element;
pile P=NULL;

void empiler(int x, pile P)
{
    Element* e;
    e = malloc (sizeof(*e));
    e->premier = x;
    e->suivant = P;
}
```

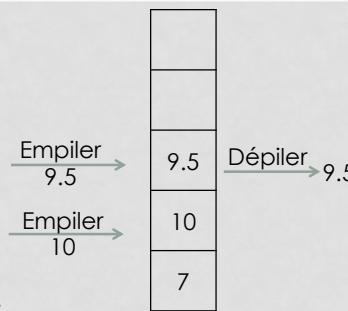


PILE (LIFO : LAST IN FIRST OUT)

```

typedef struct Element* pile;
typedef struct Element
{
    int premier;
    pile suivant;
} Element;
pile P=NULL;
int depiler(pile P) {
    if(est_vide(P)) {
        printf ("La liste est vide !\n");
        exit(EXIT_FAILURE);
    }
    Element* e = P;
    int valeur = e->valeur;
    P = P->suivant;
    free(e);
    return valeur;
}

```



FILE (FIFO : FIRST IN FIRST OUT)

```

typedef struct Element* file;
typedef struct Element
{
    int premier;
    pile suivant;
} Element;
file F=NULL;
void emfiler(int x, file F)
{
    Element *e, *eActuel = F;
    e = malloc (sizeof(*e));
    e->premier = x;
    e->suivant = NULL;
    while (eActuel->suivant != NULL)
        eActuel = eActuel->suivant;
    eActuel->suivant = e;
}

```

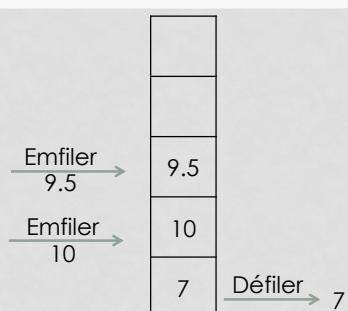


FILE (FIFO : FIRST IN FIRST OUT)

```

typedef struct Element* pile;
typedef struct Element
{
    int premier;
    pile suivant;
} Element;
pile P=NULL;
int defiler(file F) {
    if(est_vide(F)) {
        printf ("La liste est vide !\n");
        exit(EXIT_FAILURE);
    }
    Element* e = F;
    int valeur = e->valeur;
    F = F->suivant;
    free(e);
    return valeur;
}

```

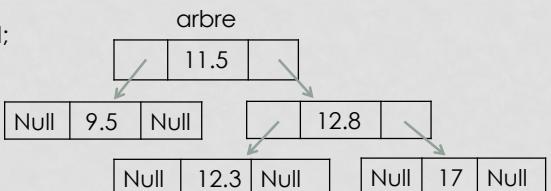


GRAPHE : ARBRE

```

typedef struct Nœud Nœud;
struct Nœud {
    float valeur ;
    Nœud *gauche ;
    Nœud *droit;
};
typedef struct Nœud* Arbre;
Nœud* créer_nœud( float valeur)
{
    Nœud* noeud = (Nœud*)malloc (sizeof(Nœud));
    noeud->valeur = valeur;
    noeud->gauche = NULL;
    noeud->droit = NULL;
    return noeud;
}

```

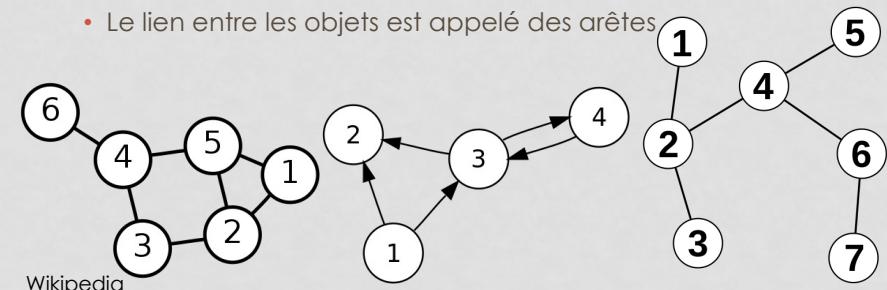


CONTENUE DU COURS

- Rappels
 - Fonctions et procédures
 - Structures de données
 - Enregistrements, tableaux, liste enchainé, Pile(FIFO), file(FILO), ...
- Compilation avec Makefile/Cmake et la bibliothèque SDL
- Gestion de mémoire
 - Représentation de mémoire
 - Pointeurs et allocation dynamique
- Gestion de fichiers
 - Lecture et écriture
- Gestion des entrées et sorties
 - Ecran, souris et clavier
- Directives au préprocesseur
- Structures de données et **algorithmes de graphes**
 - Parcours de graphe, arbre couvrant, ...

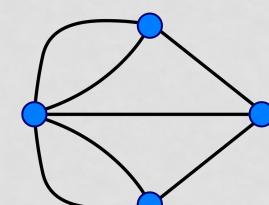
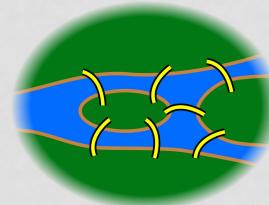
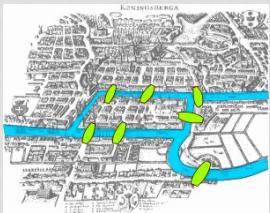
THÉORIE DES GRAPHES

- La discipline mathématique et informatique qui étudie les graphes
- Graphe est un modèle abstrait de réseaux reliant entre les objets
 - Chaque objet est représenté par un nœud
 - Le lien entre les objets est appelé des arêtes



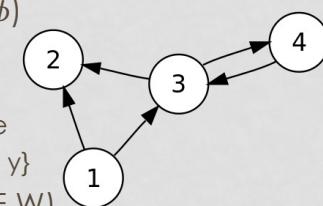
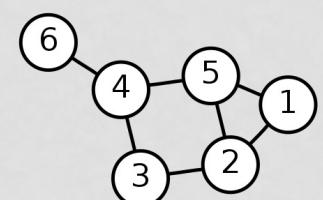
ORIGINES DE LA THÉORIE DE GRAPHES

- Mathématicien suisse Leonhard Euler
- Présenté à l'Académie de Saint-Pétersbourg en 1735 et publié en 1741
- Problème des sept ponts de Königsberg
 - Le problème consistait à trouver une chemin partant d'un point donné et revenant à ce point en passant une fois et une seule par chacun des sept ponts de Königsberg.



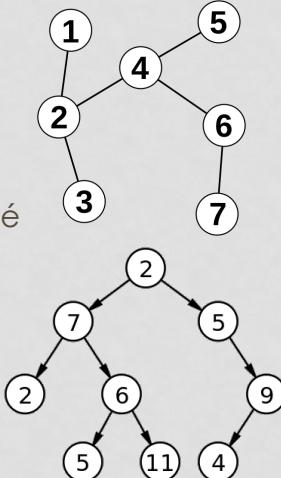
TYPES DE GRAPHE

- Graphe : $G=(V,E)$
 - V un ensemble de sommets
 - E un ensemble d'arêtes
 - $E = \{ (x,y) \mid (x, y) \in V^2 \}$
- Graphe orienté : $G=(V,E,\phi)$
 - V un ensemble de sommets
 - E un ensemble d'arêtes
 - ϕ est la fonction d'incidence
 - $\phi : E \rightarrow \{(x, y) \mid (x, y) \in V^2 \wedge x \neq y\}$
- Graphe pondéré : $G=(V,E,W)$

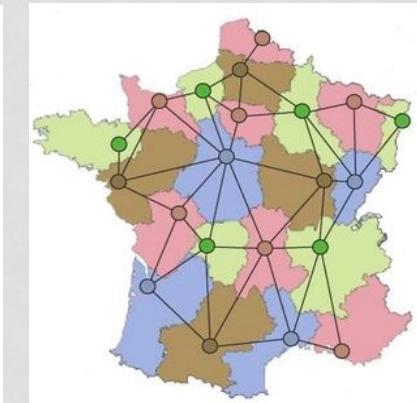


TYPES DE GRAPHE

- Arbre est un graphe non orienté, acyclique et connexe
 - Feuilles = sommets avec un seul arrêt
 - Sommets internes = autres
- Arbre binaire est un graphe orienté acyclique et connexe
 - Chaque nœud a au plus deux arrêts
 - Chaque nœud a un unique parent défini et au plus deux fils
 - La racine est le nœud sans parent
- Arbre binaire trié, arbre n-aire, ...



PROBLÈME DE QUATRE COULEURS



PROBLÈME DE QUATRE COULEURS

- Cartographe anglais Francis Guthrie
 - En 1852, il remarque qu'il lui suffit de quatre couleurs pour colorer la carte des cantons d'Angleterre
- Théorème de 4 couleurs
 - Il est possible de colorier avec quatre couleurs une carte géographique sans que deux régions voisines aient la même couleur.
- 1960-1970, Heinrich Heesch s'intéresse à la possibilité de prouver informatiquement
- 1976 Appel et Haken affirment le théorème
- 2005 Georges Gonthier et Benjamin ont formalisée une version entièrement qui permet à un ordinateur de vérifier le théorème des quatre couleurs



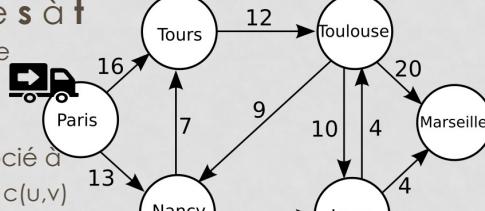
PROBLÈME DU CHEMIN OPTIMAL

- Chemin entre deux sommets sur un graphe
 - Plus court
 - Plus rapide
 - Moins de changement
 - Éviter une station
- Poids associés
 - Distance
 - Densité/fluidité
 - Vitesse
- Graphe pondéré
 - Sommet pondéré
 - Arrête pondéré



PROBLÈME DU FLOT MAXIMUM

- Un réseau de flots de s à t
 - est un graphe sans boucle
 - comporte une source
 - s et un puit (target) t
 - Chaque arc (u,v) est associé à
 - une capacité maximum $c(u,v)$
 - un flot $f(u,v)$
 - Graphe pondéré et orienté
- Problème de flot maximum consiste à trouver, dans un réseau de flots, un flot réalisable depuis une source unique et vers un puit unique qui soit maximum
 - La valeur du flot est défini par la fonction $|f| = \sum f(s,u) = \sum f(v,t)$
- Contraints sur le réseau de flots :
 - Constraint de capacité : Pour chaque arc (u,v) , $f(u,v) \leq c(u,v)$
 - Conservation du flot : Pour chaque sommet w , $\sum f(w,u) = \sum f(u,w)$
- Variances : plusieurs sources ou/et puits, capacité du sommet,...

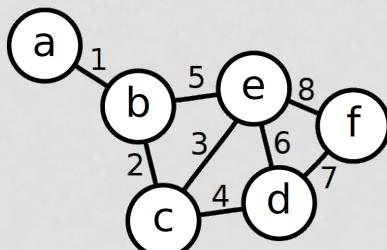


APPLICATIONS DE GRAPHES

- Modélisation par les graphes :
 - réseaux de transports
 - routage dans les circuits VLSI
 - un réseau GSM pour les opérateurs mobiles
- Problèmes classiques sur les graphes :
 - recherche de plus courts chemins
 - ordonnancement de tâches
 - problème de recherche opérationnelle
 - problèmes de flots maximum/minimum
 - problème de l'arbre recouvrant

REPRÉSENTATION DES GRAPHES

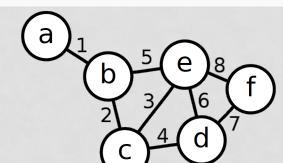
- Un graphe G est un triplet (V, E, W) où
 - V est un ensemble non vide de sommets (vertices)
 - E est un ensemble d'arêtes (edges) pour un paire de sommets
 - W est un ensemble de poids associés aux arêtes
- Exemple : Un graphe non orienté $G=(V,E, W)$ défini par
 - $V = \{a, b, c, d, e, f\}$
 - $E = \{\{a, b\}, \{b, c\}, \{c, e\}, \{c, d\}, \{b, e\}, \{d, e\}, \{d, f\}, \{e, f\}\}$
 - $W = \{1, 2, \dots, 8\}$



REPRÉSENTATION DES GRAPHES

- Le graphe $G=(V,E)$:
 - $V = \{a, b, c, d, e, f\}$
 - $E = \{\{a, b\}, \{b, c\}, \{c, e\}, \{c, d\}, \{b, e\}, \{d, e\}, \{d, f\}, \{e, f\}\}$
- Matrice orienté et pondéré

	[a]	[b]	[c]	[d]	[e]	[f]		[a]	[b]	[c]	[d]	[e]	[f]		[a]	[b]	[c]	[d]	[e]	[f]
[a]							[1]							[a]						
[b]							[2]							[b]						
[c]							[3]							[c]						
[d]							[4]							[d]						
[e]							[5]							[e]						
[f]							[6]							[f]						
							[7]													
							[8]													



REPRÉSENTATION DES GRAPHES

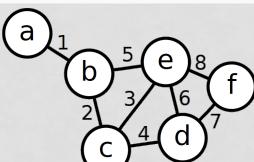
- Le graphe $G=(V,E)$:
 - $V = \{a, b, c, d, e, f\}$
 - $E = \{\{a, b\}, \{b, c\}, \{c, e\}, \{c, d\}, \{b, e\}, \{d, e\}, \{d, f\}, \{e, f\}\}$
- Matrice orienté et pondéré

- Matrice d'adjacence
 - Tableau de 2D
 - $M[i][j] = \text{True}$ s'il existe l'arrêt (i, j)

	[a]	[b]	[c]	[d]	[e]	[f]
[a]	F	T	F	F	F	F
[b]	T	F	T	F	T	F
[c]	F	T	F	T	T	F
[d]	F	F	T	F	T	T
[e]	F	T	T	T	F	T
[f]	F	F	F	T	T	F

- Matrice d'incidence
 - Tableau de 2D
 - $M[i][j] = \text{True}$ si l'arrêt i contient le sommet j

	[a]	[b]	[c]	[d]	[e]	[f]
[1]	T	T	F	F	F	F
[2]	F	T	T	F	F	F
[3]	F	F	T	F	T	F
[4]	F	F	T	T	F	F
[5]	F	T	F	F	T	F
[6]	F	F	F	T	T	F
[7]	F	F	F	T	F	T
[8]	F	F	F	F	T	T



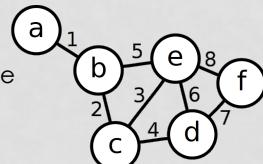
- Liste d'adjacence
 - Listes chainées
 - $[i]$: La liste des arrêts qui ont i pour origine

	[a]	b	c	d	e	f
[1]						
[2]	a		c		e	
[3]	b		d			
[4]	c		e			
[5]	b		c	d		
[6]	d	e				

STRUCTURE DE DONNÉES

- Matrice adjacent

```
int** G ; //G est le tableau représentant le graphe
FILE * pFile;
int taille;
pFile=fopen( "graph.txt" , "r");
if (pFile==NULL) perror ("Error opening file");
else {
    fscanf( pFile, "%d" , &taille) ;
    G = cree_tableau_2D (taille, taille);
    for(int i=0; i<taille ; i++) {
        for(int j=0; j<taille ; j++)
            fscanf (pFile, "%d" , &G[i][j]);
    }
    fclose (pFile);
}
```



Fichier graphe.txt

```
6
0 1 0 0 0 0
1 0 1 0 1 0
0 1 0 1 1 0
0 0 1 0 1 1
0 1 1 1 0 1
0 0 0 1 1 0
```

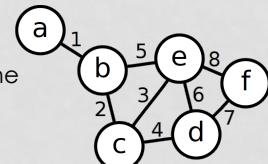
REPRÉSENTATION DES GRAPHES

	Matrice d'adjacent Matrice d'incident	Liste d'adjacence
Avantages	<ul style="list-style-type: none"> Modélisation simple Lecture/écriture simple pour chaque arrête Simplicité des algorithmes de calcul 	<ul style="list-style-type: none"> Stockage sur-mesure Examen optimal lors de parcours de graphe Modéliser les graphes non simples
Inconvénients	<ul style="list-style-type: none"> Complexité en mémoire est de $V ^2$ ou $V E$ Redondance d'info. pour les graphes non orientés Stockage et examen inutile pour les graphes peu d'arrêts Modéliser que les graphes simples 	<ul style="list-style-type: none"> Plus lente pour accéder aux sommets et arrêts Moins rapide pour des recherches Algorithmique plus complexe

STRUCTURE DE DONNÉES

- Matrice d'incident

```
int** G ; //G est le tableau représentant le graphe
FILE * pFile;
int sommet, arret;
pFile=fopen( "graph.txt" , "r");
if (pFile==NULL) perror ("Error opening file");
else {
    fscanf( pFile, "%d %d" , &sommet, &arret);
    G = cree_tableau_2D (arret, sommet);
    for(int i=0; i<arret ; i++) {
        for(int j=0; j<sommet ; j++)
            fscanf (pFile, "%d" , &G[i][j]);
    }
    fclose (pFile);
}
```



Fichier graphe.txt

```
6 8
1 1 0 0 0 0
0 1 1 0 0 0
0 0 1 0 1 0
0 0 1 1 0 0
0 1 0 0 1 0
0 0 0 1 1 0
0 0 0 1 0 1
0 0 0 0 1 1
```

STRUCTURE DE DONNÉES

- Liste d'adjacence

```
typedef struct Noeud {
    char id_noeud;
    Noeud * suivant;
}
```

```
} Noeud;
```

```
typedef struct AdjListe {
```

```
    Noeud * tete;
```

```
} AdjListe;
```

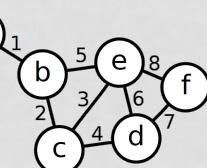
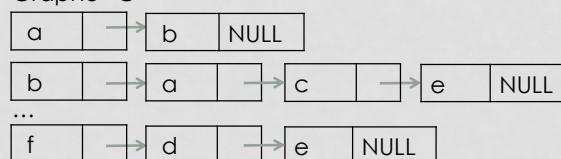
```
typedef struct Graphe {
```

```
    int nbSommet;
```

```
    AdjListe* liste;
```

```
} Graphe;
```

```
Graphe* G
```



Fichier graphe.txt

```
8
a b
b a c e
c b d e
d c e f
e b c d f
f d e
```

STRUCTURE DE DONNÉES

- Liste d'adjacence

```
Graphe* G; //G est la liste représentant le graphe
int v; FILE * pFile = fopen( "graph.txt" , "r" );
if (pFile==NULL) perror ("Error opening file");
else {
```

```
    fscanf( pFile, "%d" , &v);
```

```
    G = (Graphe*) malloc(sizeof(Graph));
```

```
    G → nbSommets = sommets;
```

```
    for(int i=0; i<sommet ; i++) {
```

```
        G → liste = (AdjListe*) malloc(v* sizeof(AdjListe));
```

```
        char c = fgetc(pFile);
```

```
        if(c == '\n') continue;
```

```
        if(c != ' ') {
```

```
            Noeud* n = (Noeud*) malloc(sizeof(Noeud));
```

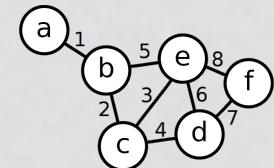
```
            n → id_noeud = c;
```

```
            n → suivant = G → liste[i].tete;
```

```
            G → liste[i].tete = n;
```

```
        }
```

```
    }
    fclose (pFile);
}
```



Fichier graphe.txt

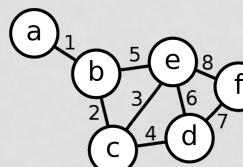
```
8
a b
b a c e
c b d e
d c e f
e b c d f
f d e
```

STRUCTURE DE DONNÉES

- Liste d'adjacence

```
Graphe* G; //G est la liste représentant le graphe
// lire et générer le graphe à partir de fichier
```

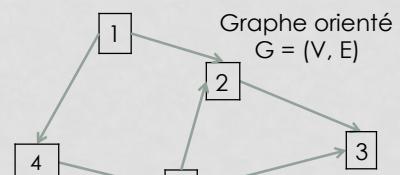
```
//afficher le graphe
for (int v = 0; v < G → nbSommet; v++) {
    Noeud* noeud = G → liste[v].tete;
    printf(" [ %c ] : ", noeud → id_noeud);
    noeud = noeud → next;
    // la liste d'adjacence du noeud
    while (noeud != NULL) {
        printf(" %c ", noeud → id_noeud);
        noeud = noeud → next;
    }
    printf("\n");
}
```



Fichier graphe.txt

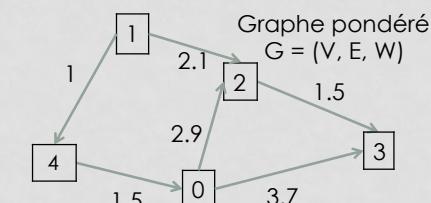
```
8
a b
b a c e
c b d e
d c e f
e b c d f
f d e
```

ALLOCATION DYNAMIQUE TABLEAU POUR LES GRAPHES



Graphe orienté
 $G = (V, E)$

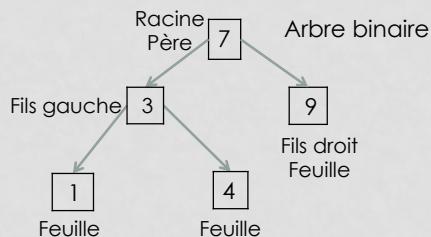
	[0]	[1]	[2]	[3]	[4]
[0]	F	F	V	V	F
[1]	F	F	V	F	V
[2]	F	F	F	V	F
[3]	F	F	F	F	F
[4]	V	F	F	F	F



Graphe pondéré
 $G = (V, E, W)$

	[0]	[1]	[2]	[3]	[4]
[0]	0	0	2.9	3.7	0
[1]	0	0	2.1	0	1
[2]	0	0	0	1.5	0
[3]	0	0	0	0	0
[4]	1.5	0	0	0	0

ALLOCATION DYNAMIQUE TABLEAU POUR LES GRAPHS



	[0,7]	[1,3]	[2,9]	[3,1]	[4,4]
[0,7]	F	T	T	F	F
[1,3]	F	F	F	T	T
[2,9]	F	F	F	F	F
[3,1]	F	F	F	F	F
[4,7]	F	F	F	F	F

- Un arbre A=(V,E) : Un graphe sans cycle
 - V : ensemble de sommets, #V = n
 - E : ensemble des arrêts, #E = n-1
- Problème de recherche, de compression, le Huffman coding

ALLOCATION DYNAMIQUE STRUCTURE DE GRAPHE : ARBRE

- Une structure récursive qui permet de faire référence à une autre structure de même type
 - Exemples : listes chainées, arbres binaires, ...

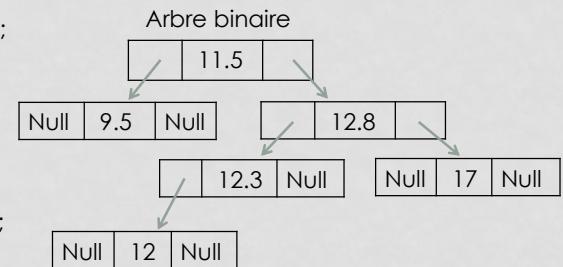
typedef struct Nœud Nœud;

```

struct Nœud {
    float valeur;
    Nœud *gauche;
    Nœud *droit;
};
  
```

typedef struct Nœud* Arbre;

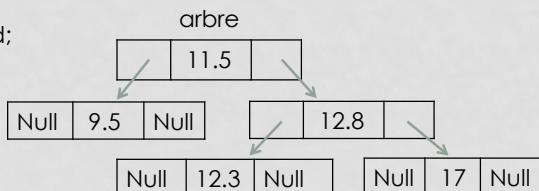
Arbre arbre;



ALLOCATION DYNAMIQUE STRUCTURE DE GRAPHE : ARBRE

```

typedef struct Nœud Nœud;
struct Nœud {
    float valeur;
    Nœud *gauche;
    Nœud *droit;
};
typedef struct Nœud* Arbre;
Nœud* créer_noeud( float valeur)
{
    Nœud* noeud = (Nœud*)malloc(sizeof(Nœud));
    noeud->valeur = valeur;
    noeud->gauche = NULL;
    noeud->droit = NULL;
    return noeud;
}
  
```



ALLOCATION DYNAMIQUE STRUCTURE DE GRAPHE : ARBRE

typedef struct Nœud Nœud;

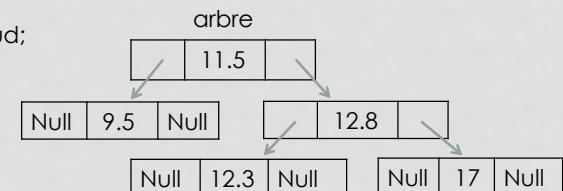
```

struct Nœud {
    float valeur;
    Nœud *gauche;
    Nœud *droit;
};
  
```

typedef struct Nœud* Arbre;

```

void insérer_element (Nœud* abré, float valeur)
{
    if (est_vide(abré)) abré = créer_noeud(valeur);
    else {
        if (abré->valeur > valeur) { //On regarde à gauche
            if(abré->gauche == NULL) abré->gauche = créer_noeud(valeur);
            else insérer_element(abré->gauche, valeur);
        }
        else { /* on faire la même chose à droite */ }
    }
}
  
```



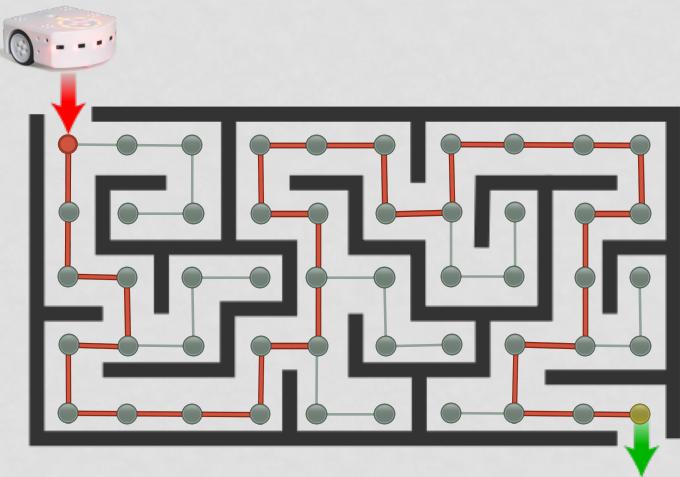
ALLOCATION DYNAMIQUE STRUCTURE DE GRAPHE : ARBRE

```

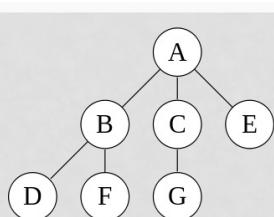
typedef struct Nœud Nœud;
struct Nœud {
    float valeur ;
    Nœud *gauche ;
    Nœud *droit;
};
typedef struct Nœud* Arbre;
Nœud* chercher_element(Nœud* abre, float valeur)
{
    if (est_vide(arbre)) return NULL
    if (arbre->valeur == valeur) return arbre;
    if (arbre->valeur > valeur)
        return chercher_element(abre->gauche, valeur);
    else
        return chercher_element(abre->droit, valeur);
}

```

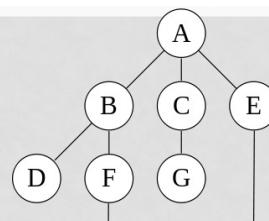
PARCOURS DE GRAPHE



PARCOURS DE GRAPHE EN PROFONDEUR



[A, B, D, F, C, G, E]

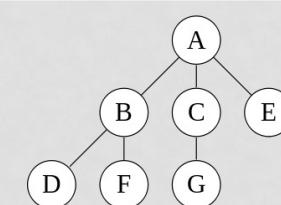


[A, B, D, F, E, A, B, D, F, E, ...]

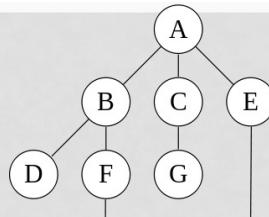
1. Parcours_profondeur (graphe G, sommet **s**)
2. marquer le sommet **s** comme passé
3. afficher(**s**)
4. Pour tous sommets **u** voisin de **s** faire
5. Si **u** n'est pas marqué alors
6. Parcours_profondeur (G, **u**)

(Wikipedia)

PARCOURS DE GRAPHE EN LARGEUR



[A, B, C, E, D, F, G]

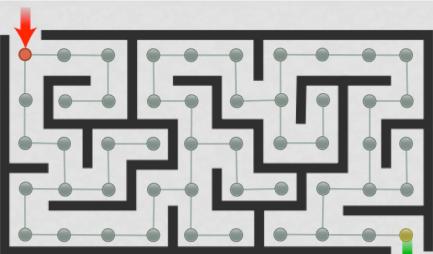


[A, B, C, E, D, F, G]

1. Parcours_largeur (graphe G, sommet **s**)
2. **F** = FILE()
3. **F**.enfiler(**s**)
4. marquer le sommet **s** comme passé
5. Tant que **F** n'est pas vide
6. **s** = **F**.defiler()
7. afficher(**s**)
8. Pour tous sommets **u** voisin de **s** faire
9. Si **u** n'est pas marqué alors
10. **F**.enfiler (**u**)
11. marquer le sommet **u** comme passé

(Wikipedia)

CHERCHE DE CHEMIN

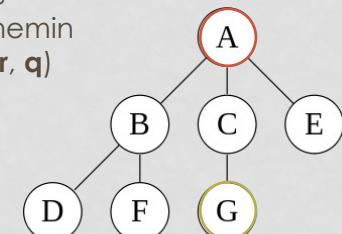


1. Commencer par la case entrée du labyrinthe
2. Si la case courante est la sortie, alors c'est terminé !
3. Sinon, marquer la case comme passé
4. Pour chaque case C atteignable, si la case C :
 - a. n'est pas marquée
 - i. et n'est pas bloqué, l'explorer en recommençant au point 2
 - ii. et bloqué, on revient arrière sur une autre branche et recommençant au point 2
 - b. est marqué, on passe à une autre case qui n'a pas marqué et recommençant au point 2

CHERCHE DE CHEMIN

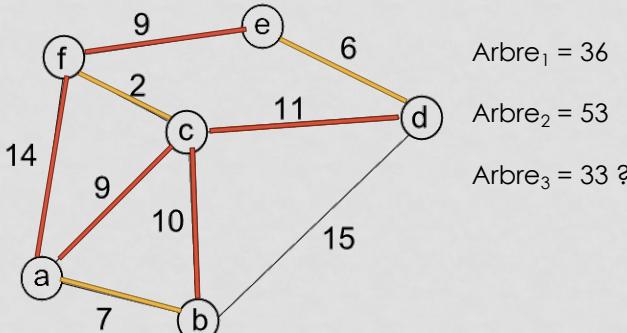
Recherche dans le graphe G un chemin de **p** à **q**
 Chemin (graphe G, sommet **p**, sommet **q**) : booléen

1. Si **p** == **q** alors
 Retourner Vrai
2. marquer le sommet **p** comme passé
3. Pour tout sommet **r** voisin du sommet **p** faire
 5. Si **r** n'est pas marqué alors
 6. $\Pi[r] \leftarrow p$ // Suivi du chemin
 7. trouvé \leftarrow Chemin(G, **r**, **q**)
 8. Si trouvé == Vrai alors
 Retourner Vrai
 - 9.
 10. Retourner Faux



ARBRES COUVRANTS MINIMAUX

- Problème : Étant donné un graphe pondéré, trouver un sous-ensemble d'arêtes formant un arbre sur l'ensemble des sommets du graphe initial, et tel que la somme des poids de ces arêtes soit minimale



ARBRES COUVRANTS MINIMAUX

• Algorithme de Kruskal

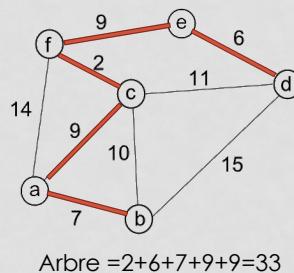
Entrées : **G=(V,E,W)** un graphe pondéré
 ArbreRecouvrementMinimal (graphe G) : arbre

1. $A \leftarrow \emptyset$
2. nombre_sommet $\leftarrow 0$
3. $L \leftarrow$ trier les arêtes de **E** par poids croissants
4. Tant que nombre_sommet < |V| faire
 5. **e** \leftarrow arête dans L de poids minimum
 6. retirer **e** dans L
 7. Si (A U {**e**}) ne contient pas de cycle alors
 8. $A \leftarrow A \cup \{e\}$
 9. nombre_sommet \leftarrow nombre_sommet + 1
 10. Retourner A

(Wikipedia)

ALGORITHME DE KRUSKAL

3. $L \leftarrow$ trier les arêtes de E par poids croissants
4. Tant que $\text{nombre_sommet} < |V|$ faire
5. $e \leftarrow$ arête dans L de poids minimum
6. retirer e dans L
7. si ($A \cup \{e\}$) ne contient pas de cycle alors
8. $A \leftarrow A \cup \{e\}$
9. $\text{nombre_sommet} \leftarrow \text{nombre_sommet} + 1$



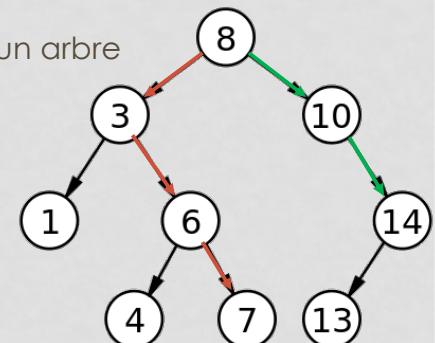
- $L = \{[cf]_2, [ed]_6, [ab]_7, [ac]_9, [fe]_9, [bc]_{10}, [cd]_{10}, [af]_{14}, [bd]_{15}\}$
- $A = \{[cf]_2, [ed]_6, [ab]_7, [ac]_9, [fe]_9\}$

RECHERCHE AVEC ARBRE BINAIRE TRIÉ

- Arbre binaire trié est un arbre binaire tel que le fils à gauche contient une valeur plus petit que celle du parent et le fils droit contient une valeur plus grande que celle du parent
- Problème : étant donnée un arbre binaire trié A , chercher un élément x dans A

$x = 7 \rightarrow \checkmark$

$x = 15 \rightarrow \times$



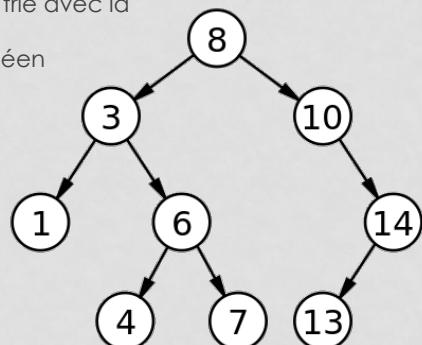
RECHERCHE AVEC ARBRE BINAIRE TRIÉ

- Méthode de recherche

Entrées : $A=(V,E)$ un arbre binaire trié avec la racine r et x élément à chercher

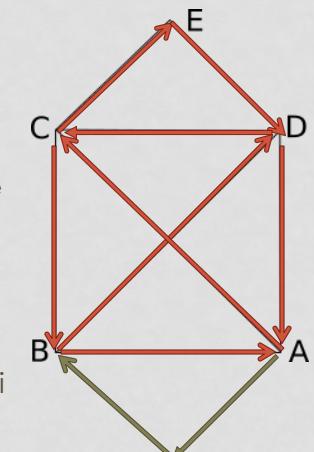
Recherche (arbre A , entier x) : booléen

1. $v \leftarrow A.\text{racine}$
2. Tant que $v \neq \text{NULL}$ faire
3. Si $v.\text{valeur} == x$ alors
4. Retourner Vrai
5. Si $v.\text{valeur} < x$ alors
6. $v \leftarrow v.\text{gauche}$
7. Sinon
8. $v \leftarrow v.\text{droite}$
9. Retourner Faux



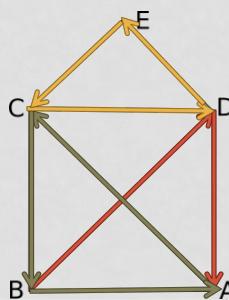
CHEMINS EULÉRIENS

- Problème : un chemin Eulérien d'un graphe G est un sous-graphe de G contenant une fois et une seule chaque arête de G
 - Un chemin Eulérien de G qui est un cycle, alors c'est un cycle Eulérien
 - Un tel chemin/cycle a alors la propriété qu'il correspond à un dessin qu'on peut tracer sans lever le crayon
- Théorème d'Euler : Si pour tout sommet du graphe G , $d(i)$ est pair, alors G admet un cycle Eulérien
- Théorème : Un chemin Eulérien distincte d'un cycle si et seulement si le nombre de sommets de degré impair est égal à 2.



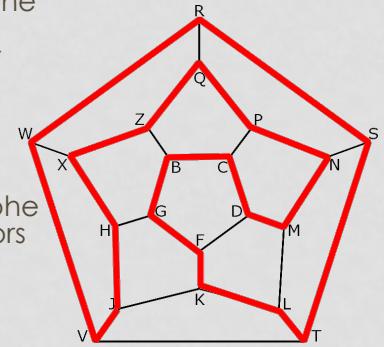
CHEMINS EULÉRIENS

- Algorithme pour chercher un chemin Eulérien
- nombre_impair \leftarrow le nombre de sommets de degré impair
 - Si nombre_impair == 0 alors
 - construire un cycle partant de n'importe sommet
 - Sinon si nombre_impair == 2 alors
 - construire un chemin entre les deux sommets en de degré impair **x** et **y**
 - Sinon alors il n'y a pas de chemin eulérien
-
- $R \leftarrow (x=y_1, \dots, y=y_n)$ // R est un chemin de **x** à **y**
 - supprimer les arêtes de R dans **G**
 - $R \leftarrow \emptyset$
 - Pour i allant de 1 à n faire
 - Si les arrêts de y_i ne sont pas tous visités alors
 - $C \leftarrow \text{cycle_Eulerien}(G, y_i)$
 - $R \leftarrow R \cup C$
 - supprimer les arêtes de R dans **G**



CYCLES HAMILTONIENS

- Problème : un cycle Hamiltonien d'un graphe G est un sous-graphe de G contenant au moins un cycle passant une seule fois par tous les sommets de G
- La recherche d'un cycle Hamiltonien est NP-complet
- Critère de Ore : Si pour tout couple de sommets (i,j) du graphe G non adjacent, $d(i)+d(j)\geq N$, alors G est Hamiltonien
- Critère de Dirac : Si pour tout sommet du graphe G, $d(i)\geq N/2$, alors G est Hamiltonien



COLORATION DE GRAPHES

- Problème : La coloration de graphe consiste à attribuer une couleur à chacun de ses sommets de manière que deux sommets reliés par une arête soient de couleur différente.
 - Problème plus général : chercher le nombre minimal de couleurs, appelé **nombre chromatique**. (NP-complet)
- La méthode simple est de parcourir le graphe et attribuer une couleur autre que ses voisins en essayant d'utiliser le moins de couleur possible
 - Idée : on utilise une nouvelle couleur seulement si c'est vraiment nécessaire

