Structure de données

https://sleek-think.ovh/enseignement

Jehan-Antoine Vayssade

Compétence visées

- Compréhension des différents types de structures de données
- Conception et mise en oeuvre en C
- Analyser et comprendre la complexité temporelle et spatial
- Comprendre les optimisations mémoire (cache miss)
- Savoir utiliser la bonne structure de donnée pour optimiser vos algorithms
- Utilisation avancée des pointeurs et de la gestion de la mémoire

Programme

- Niveau 0 : Présentation et notion de complexité
- Niveau 1 : Retour sur le tableaux et listes (dynamique vs statique)
  - Listes chainées simple, double, circulaires (push/pop/find)
- Niveau 2 : Piles et files (LIFO / FIFO) et cas d'utilisation
- Niveau 3 : Structures de données hiérarchiques (arbre binnaire, arbre avl, graph, parcours d'arbre)
- Niveau 4 : Algorithme de trie et de recherche (dichotomique, quicksort, boustrophedon, …)

Notes

- Moyenne des TPs coef 1
  - ~1 TP par section
- Exam coef 1

Convention d’ecriture

- Style consistant !
  - Vous pouvez utiliser `clang-format`
  - Nommer correctement les fonction et variables (semantique)
  - Indentation (bloque)
  - Aérer les lignes du programme
  - Facilite la lecture !!!
- Pas de variables globales
- Commenter vos programmes
  - Pas de commentaire initile
- Attention sanction sur les notes de TP
- Example https://github.com/MaJerle/c-code-style

-2

```
#include<stdio.h>
#define MAX 1000
int a,b,c;float d;
char e[MAX];void F(int x){
if(x>0){printf("Positive");
}else{if(x<0){
printf("Negative");}else{printf("Zero");}}}
int main()
{
a=10;b=20;
c=a+b;
d=3.14159;
// This is a useless comment
printf("Hello World!");F(c);
for(int i=0;i<MAX;i++){e[i]='A';}
    if(d>3)
{
        printf("d is greater than 3");
    }
    else
        {
        printf("d is not greater than 3");
        }
return 0;
}
```

Niveau 0

Comprendre la complexité

Introduction

Pourquoi est-elle essentielle dans le développement logiciel ?

Objectifs de cette présentation :

- Définir la complexité des algorithmes
- Expliquer les notations asymptotiques (O, Ω, Θ)
- Discuter de l’importance des constantes dans ces notations
- Présenter des exemples pratiques en pseudo-C

Pourquoi est-ce important ?

- Optimisation des performances (CPU vs RAM)
- Provisionné vos ressources (contrainte matériel)
- Amélioration de la qualité du code

Définition de la complexité des algorithmes

Mesurer l'Efficacité des Programmes

La complexité d'un algorithme est une mesure de la quantifir les ressources nécessite pour traiter une donnée. Elle nous permet de prédire comment l'algorithme se comportera lorsque la taille des données augmente.

Types de Complexité :

- **Complexité Temporelle** : Temps nécessaire pour exécuter l'algorithme (CPU).
- **Complexité Spatiale** : Espace mémoire nécessaire pour exécuter l'algorithme (RAM).

Pourquoi est-elle importante ?

- **Optimisation des performances** : Améliorer la rapidité et l'efficacité.
- **Évolutivité** : Prévoir comment l'algorithme se comportera avec des données plus volumineuses.
- **Qualité du code** : Écrire des programmes plus efficaces et maintenables.

Similariter

- Pour faire simple compter le nombre d'opération quand : $\lim_{n \to +\infty} f(n)$

Exemple simple :

Supposons un algorithme de recherche dans une liste de n éléments.

Voici un pseudo-algo qui vérifie un par un les éléments de la liste pour le trouver

```
int recherche(int *array, size_t size, int value) {
    int index = -1;
    for (size_t i=0; i<size; ++i)
        if (array[i] == value)
            index = i;
    return index;
}
```

Cette algorithm a une complexité temporel lineaire, dans tout les cas le temps d'éxécution d'épendra de la taille de `@array` .

Notations asymptotiques

Les notations asymptotiques sont utilisées pour décrire la complexité d'un algorithme en fonction de la taille de l'entrée. Elles nous aident à prédire comment l'algorithme se comportera lorsque la taille des données augmente.

Big O

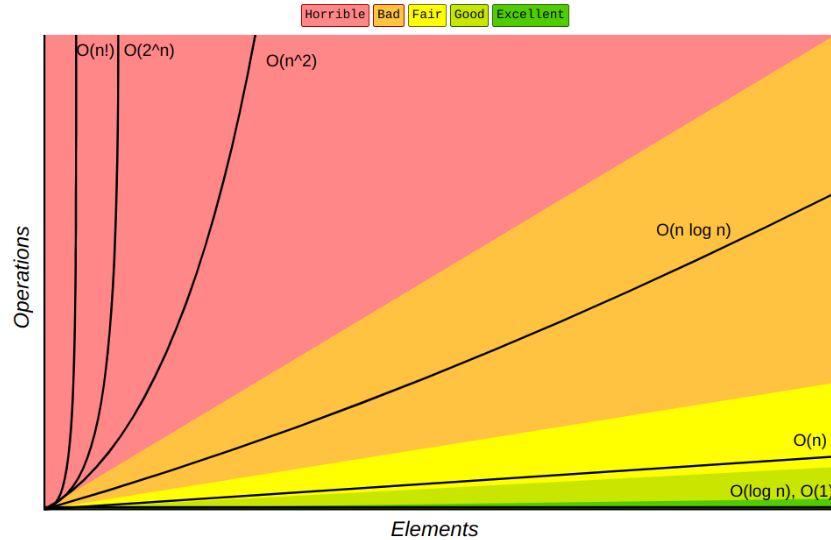
- **Définition :** Limite supérieure. C'est la pire des situations.
- **Interprétation :** L'algorithme ne prendra jamais plus de temps que $O(n)$ pour une liste de n éléments.

Big Ω (parfois minus o)

- **Définition :** Limite inférieure. C'est la meilleure des situations.
- **Interprétation :** L'algorithme prendra au minimum $\Omega(n)$ temps pour une liste de n éléments.

Big Θ

- **Définition :** Limite exacte à la complexité d'un algorithme. Quand $\mathbf{O(f)=\Omega(f)}$
- **Interprétation :** L'algorithme prend exactement $\Theta(n)$ temps pour une liste de n éléments.



- 🚀 Complexité constante **O(1)**
- 🚂 Complexité logarithmique **O(log(n))**
- 🚗 Complexité linéaire **O(n)**
- 📈 Complexité quasi-linéaire **O(n · log(n))**
- ⚠️ Complexité quadratique **$O(n^2)$**
- 🚧 Complexité exponentielle **O(2n)**
- 🚨 Complexité factorielle **O(n!)**

Notation asymptotiques et approximation

Dans la notation asymptotique, on oublie les termes constants

- $O(2n) = O(n)$: Les constances sont ignorées, donc multiplier par 2 ne change "pas" la complexité.
- $O(3n^2) = O(n^2)$: De même, les constantes devant les puissances sont ignorées.
- $O(n + 10) = O(n)$: Les termes constants sont négligés par rapport aux termes variables.
- $O(n^2 + n) = O(n^2)$: Dans les polynômes, on ne garde que le terme dominant.

Pourquoi ignorer les constantes ?

- **Échelle de grandeur :** Lorsque la taille des données augmente, les constantes deviennent négligeables par rapport à la croissance globale.
- **Modélisation simplifiée :** Ignorer les constantes simplifie l’analyse et permet de se concentrer sur la tendance générale.

Attention

> Ses notions sont a savoir et utilisées dans les tests de recrutement, mais c’est fondamentalement incorrect !

Pourquoi les constantes sont-elles importantes ?

- **Performances réelles :** Dans la pratique, les constantes peuvent avoir un impact significatif sur les performances, surtout si elles représentent des opérations complexes ou des accès mémoire coûteux.
- **Optimisation :** Ignorer les constantes peut conduire à négliger des opportunités d’optimisation importantes. Par exemple, un algorithme avec une constante plus petite mais une complexité similaire est plus rapide en pratique.
- **Cas réels :** Dans de nombreux cas réels, les tailles des données ne sont pas toujours très grandes, et les constantes peuvent donc jouer un rôle crucial dans la performance globale.
- **Taille des données :** Si la notation Big O ce concentre sur de large échelles de donnée, en pratique vous connaitrez presque toujours les tailles. Elles seront presque toujours petites dans ce cas un algorithme en O(n) peut être meilleur qu’un algorithm en O(log(n))’

What Big-O notation ACTUALLY tells you, and how I almost failed my Google Inte...

Partager

Big-O

Regarder sur YouTube

Exemples de complexité en $\Theta(1)$

–

Toute opération qui accede directement a une donnée:

```
int tableau[1000];
int index = 5;
int valeur = tableau[index];
```

ou effectue des calcules

```
bool estPair(int nombre) {
  return nombre % 2 == 0;
}
```

- Ceci n’est pas vrais pour certaines structures de données complexe (hash map, bin tree, etc) "caché"
- Quelle complexité est caché pour la fonction estPair ?

Quelle est la complexité de ce code ?

- Complexité spatial ?
- Complexité temporel ?

```
int x = 3;
int n = 10;
int result = pow(x, n) * sqrt(x, n);
```

- **A** : $\Theta(1)$
- **B** : $\Theta(2)$
- **C** : On ne sait pas ?
- **D** : Expliquer

Exemples de complexité en Θ(n)

```c
double pow(double base, int exponent) {
  double result = 1.0;
  for (int i = 0; i < exponent; i++) {
    result *= base;
  }
  return result;
}
```

```c
int recherche(int *array, size_t size, int value) {
    int index = -1;
    for (size_t i=0; i<size; ++i)
        if (array[i] == value)
            index = i;
    return index;
}
```

Quelle est la complexité de ce code ?

```
int recherche(int *array, size_t size, int value) {
  for (size_t i=0; i<size; ++i)
    if (array[i] == value)
      return i;
  return -1;
}
```

- Complexité spatial ?
  - **A**: $O(1)$
  - **B**: $\Theta(1)$
  - **C**: $\Omega(1)$
- Complexité temporel ?
  - **A**: $O(n)$
  - **B**: $\Theta(n)$
  - **C**: $\Omega(1)$ et $O(n)$

Exemples de complexité en Θ(log(n))

```
double pow(double base, int exponent) {
    double result = 1.0;

    while (exponent > 0) {
        if (exponent % 2 == 1)
            result *= base;
        exponent /= 2;
        base *= base;
    }

    return result;
}
```

Exemples de complexité en $\Theta(\log(n))$

```c
double sqrt(double x) {
    if (x < 0)
        return NAN;
    else if (x == 0 || x == 1)
        return x;

    double guess = x / 2.0;
    double precision = 0.000001;

    while (1) {
        double betterGuess = (guess + x / guess) / 2.0;
        if (fabs(guess - betterGuess) < precision)
            return betterGuess;
        guess = betterGuess;
    }
}
```

Peut-on faire mieux ?

John Carmack et Quake III

```
float fast_sqrt(float number) {
    long i;
    float x2, y;
    const float threehalfs = 1.5F;

    y = number;
    i = *(long*)&y;
    i = 0x5f3759df - (i >> 1);
    y = *(float*)&i;

    y = y * (threehalfs - (x2 = number * 0.5F) * y * y);
    y = y * (threehalfs - (x2 = number * 0.5F) * y * y);

    return 1/y;
}
```

`https://en.wikipedia.org/wiki/Fast_inverse_square_root`

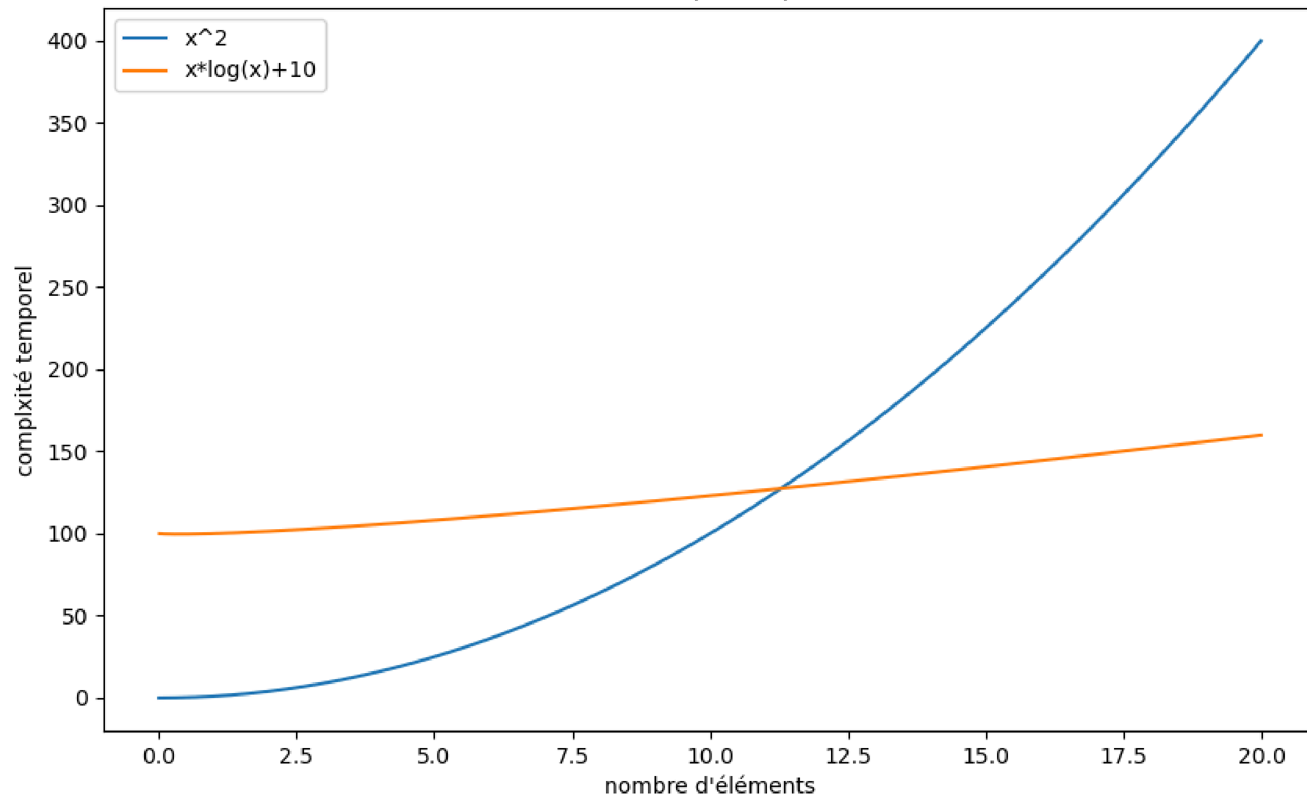
Exemples de complexité en Θ(n^2)

```
double pow(double base, int exponent) {
    double result = 1.0;

    for (int i = 0; i < exponent; i++)
        for (int j = 0; j < exponent; j++)
            if (i == j)
                result *= base;

    return result;
}
```

> Attention, le compilateur passe par là et fait aussi des optimisations ! (-Ofast -O3 -Og) ;)


Qui est le plus rapide ?
x^2
x*log(x)+10
complxité temporel
nombre d'éléments
0
50
100
150
200
250
300
350
400
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

Niveau 1

Tableaux et listes

Tableau statique

Les tableaux statiques sont des tableaux dont la taille est fixée au moment de la compilation.

- Ont peu calculer la taille des tableaux statique à l'aide de `sizeof(localArray)/sizeof(*localArray)`
- Attention l'espace de stockage diffère et peu dans certain cas être `read only`

```c
// stockage dans .text / might be read only
static const char DATA[] = "SOME DATA";
// stockage dans .data
int globalArray[10] = {1, 2, 3, 4, 5, 0, 0, 0, 0, 0};
// stockage dans .bss
int globalUninitializedArray[10];
// stockage dans .rodata ! / read only
const int globalConstArray[] = {1, 2, 3, 4, 5};
int data[] __attribute__((section("rodata"))) = {1, 2, 3, 4, 5};

int main() {
    // stockage dans .data
    static int localStaticArray[10] = {1, 2, 3, 4, 5, 0, 0, 0, 0, 0};
    // stockage dans .bss
    static int localStaticUninitializedArray[10];
    // dans une fonction, stockage dans la stack
    int localArray[10];
}
```

Tableau statique

Sur une architecture 64bits : Que fait le code suivant ?

```c
#include <stdio.h>

void afficheTaille(int *array) {
  printf("%ld\n", sizeof(array));
  printf("%ld\n", sizeof(array)/sizeof(*array));
}

int main() {
  int localArray[10];
  printf("%ld\n", sizeof(localArray));
  printf("%ld\n", sizeof(localArray)/sizeof(*localArray));
  afficheTaille(localArray);
}
```

Tableau dynamique

Les tableaux dynamiques sont des tableaux dont la taille est déterminée au moment de l'exécution.

- La mémoire des tableaux dynamiques est allouée à l'aide de fonctions d'allocation
- `malloc()`, `calloc()` et `realloc()`.
- Attention aux fuite mémoire, ne pas oublié `free()`
- Il faut stocker la taille du segment associé, ou avoir une méthode de détection

```c
unsigned int size = 10 * sizeof(int);
// stockage dans la heap
int *array = malloc(size);
// attention, malloc may fail (return NULL)
...
unsigned int new_size = 100 * sizeof(int);
int* temp = realloc(array, new_size);
// attention, realloc may fail (return NULL, @array still valid)
array = temp;
...
free(array);
```

Liste chainées

- **Concept :**
  - On associe une donnée avec un ou plusieurs pointeurs
- **Avantages :**
  - Insertion et suppression efficaces : En modifiant les pointeurs des nœuds adjacents.
  - Utilisation "efficace" de la mémoire : Seule la mémoire "nécessaire" est allouée
- **Inconvénients :**
  - Recherche très lente : Doit parcourir la liste séquentiellement (mémoire non contigue).
  - Peu d'optimisation possible

Pensé a utiliser une structure pour géré votre liste :

```c
typedef struct LinkedList {
    struct Node* head;
    struct Node* tail;
    // etc
} LinkedList;
```

Liste chainées simple

- **Définition :** Chaque élément (ou nœud) pointe vers le suivant.
- **Head :** Le premier élément de la list (NULL si vide)
- **Tail :** Le dernier élément de la list (NULL si vide, next = NULL)

```c
typedef struct IntNode {
    int data;
    struct IntNode* next;
} IntNode;

typedef struct UserDataNode {
    int size;
    void *data;
    struct UserDataNode* next;
} UserDataNode;
```

Liste chainées double

- **Définition :** Chaque élément (ou nœud) pointe vers le suivant et le précédent.
- **Mémoire :** Consomme plus de mémoire.
- **Head :** Le premier élément de la list (prev = NULL)
- **Tail :** Le dernier élément de la list (next = NULL)

```c
typedef struct IntNode {
    int data;
    struct IntNode* next;
    struct IntNode* prev;
} IntNode;

typedef struct UserDataNode {
    int size;
    void *data;
    struct UserDataNode* next;
    struct UserDataNode* prev;
} UserDataNode;
```

Liste chainées circulaire

- **Définition :** Il n’existe pas de `head` ou de `tail` mais plutot un `cursor`
- **Simple ou double :** A choisir en fonction des besoins
- **Prev :** Prédécésseur n’est jamais null (sauf si list vide)
- **Next :** Sucesseur n’est jamais null (sauf si list vide)

Circular Buffer

- Même principe mais dans une zone mémoire contigue
- Implémenter les deux versions

Exercice :

- Partir du code https://classroom.github.com/a/g5Lddf1f
- Ecrire un code dans un nouveau fichier (compilation séparé)
  - Les tableaux dynamique et statique (`dynamic_array.c` et `static_array.c`)
  - Les listes simple `single_linked_list.c` et double `double_linked_list.c`
  - Les listes circulaire `circular_linked_list.c`
  - **Bonnus :** Écrire un CircularBuffer (Tableaux dynamique + List Circulaire)
- Pour chaque méthode implémenter `push_front` / `push_back` / `insert(index)` / `remove(index)`
  - Ex: `push_front(SingleLinkedList*, SLL_IntNode*)`
- Pour chaque méthode expliquer les différents problèmes
  - Complexité temporel
  - Complexité spatial
  - **Bonnus :** Expliqué les problemes de cache miss associés, avec schéma
- Chercher des exemples de cas d'utilisation de ces structures de données
- Petit rapport (-2 si pas de rapport)

Problème de cache miss

- **Accès imprévisible à la mémoire :** Les nœuds sont dispersés dans la mémoire.
- **Manque de localité :** Pas d'accès aux emplacements de mémoire direct [index].
- **Augmentation du temps d'accès à la mémoire :** Les échecs de mise en cache entraînent un ralentissement des performances en raison de l'attente des données de la mémoire principale.
- **Chargement inefficace des blocs de mémoire :** Un seul nœud par bloc de cache est utilisé
- **Impact :** Ralentissement de la traversée, recherche très lente.

Niveau 2

FIFO et LIFO

Introduction à FIFO

FIFO est un principe de gestion des données où le premier élément ajouté est le premier à être traité. Cela ressemble à une file d'attente dans la vie réelle, où la personne qui arrive en premier est servie en premier.

Opération	Description
Enqueue	Ajouter un élément à la fin de la file.
Dequeue	Supprimer l'élément du début de la file.
Peek	Voir l'élément du début sans le supprimer.

Cas d’utilisation de la Pile

- Gestion de l’historique de navigation dans les navigateurs web
- Fonctionnalités "Annuler/Rétablir" dans les éditeurs de texte (2 piles)
- Appel de fonctions et gestion de la récursion (call stack)
- Évaluation d’expressions mathématiques (infixes)
- Parcours de graph en profondeur (Depth-first search)
- Analyse syntaxique : (syntaxe des programmes - parenthèses et les accolades)

Introduction à LIFO

LIFO est un principe de gestion des données où le dernier élément ajouté est le premier à être traité. Cela ressemble à une pile de livres, où le livre posé en dernier est le premier à être retiré.

Opération	Description
Push	Ajouter un élément au sommet de la pile.
Pop	Supprimer l'élément du sommet de la pile.
Peek	Voir l'élément du sommet sans le supprimer.

Cas d’utilisation de la File

- Gestion des processus dans les systèmes d’exploitation
- Gestion des impressions
- Recherche en largeur (BFS) (explorer les graphes niveau par niveau)
- Gestion des appels dans les centres d’appel
- Tampon de messages dans les systèmes de communication

Quelles sont les différences ?

Caractéristique	FIFO	LIFO
Ordre de Traitement	Premier entré, premier sorti	Dernier entré, premier sorti
Utilisations	Planification des tâches, messagerie	Appels de fonctions, annulation
Implémentation	File (Queue)	Piles (Stack)

On peut implémenter LIFO/FIFO de différentes façons :

- Avec un tableau (gestion des réallocations potentielles)
- Avec une liste chainée (problème de cache miss)

Niveau 3

Structures de données hiérarchiques

Arbres binnaire

Un arbre binaire est une structure de données où chaque nœud peut avoir au maximum deux enfants. Les arbres binaires sont utilisés pour représenter des hiérarchies et sont efficaces pour certaines opérations comme la recherche et l'insertion.

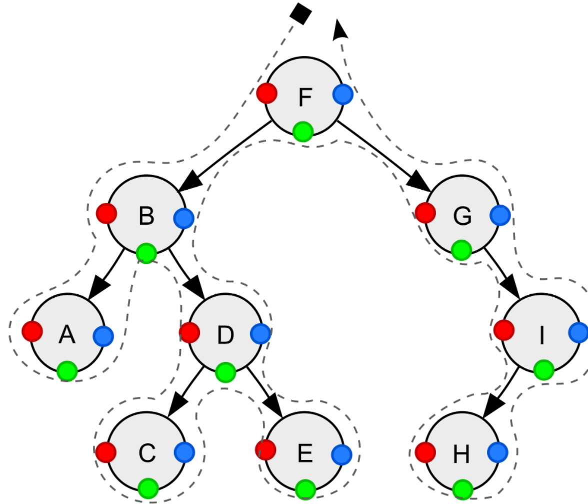
- **Maximum de deux enfants** : Chaque nœud a au plus deux enfants.
- **Structure hiérarchique** : Les nœuds sont organisés de manière hiérarchique.
- **Flexibilité** : Peuvent être utilisés pour diverses applications sans nécessiter un ordre spécifique.
- **Problème de balancement :** Largeur et profondeur variable -> change la complexité temporel

Comment peut-on écrire cette structure de données en C ?

Structure en C

```
typedef struct Node {
  void *data;
  struct Node *left, *right;
} Node;

Node node = { .data=((void*)5), .left=NULL, .right=NULL };
printf("Hello World %lld", (long long)node.data);
```



source: wikipedia

Arbre binnaire : TP

```
Node root = {
    .data = 'F',
    .left = &(Node){
      .data = 'B',
      .left = &(Node){ .data = 'A', .left = NULL, .right = NULL, },
      .right = &(Node){
        .data = 'D',
        .left = &(Node){ .data = 'C', .left = NULL, .right = NULL, },
        .right = &(Node){ .data = 'E', .left = NULL, .right = NULL, },
      },
    },
    .right = &(Node){
      .data = 'G',
      .left = NULL,
      .right = &(Node){
        .data = 'I',
        .left = &(Node){ .data = 'H', .left = NULL, .right = NULL, },
        .right = NULL,
      },
    },
};
```

Traverser un arbree binnaire (non-optimisé)

```
void inorder(Node *tree, void *user, void(*dosomethink)(Node*, void*))
{
    if(!tree) return;
    inorder(tree->left, user, dosomethink);
    dosomethink(tree, user); // A, B, C, D, E, F, G, H, I
    inorder(tree->right, user, dosomethink);
}

void preorder(Node *tree, void *user, void(*dosomethink)(Node*, void*))
{
    if(!tree) return;
    dosomethink(tree, user); // F, B, A, D, C, E, G, I, H
    preorder(tree->left, user, dosomethink);
    preorder(tree->right, user, dosomethink);
}

void postorder(Node *tree, void *user, void(*dosomethink)(Node*, void*))
{
    if(!tree) return;
    postorder(tree->left, user, dosomethink);
    postorder(tree->right, user, dosomethink);
    dosomethink(tree, user); // A, C, E, D, B, H, I, G, F
}
```

Arbre binaire de recherche

C’est un arbre binaire pour le quel on associe une fonction de comparaison a chaque noeud.

Propriété:

- toutes les clés dans le sous-arbre gauche sont inférieures à la clé du nœud
- toutes les clés dans le sous-arbre droit sont supérieures
- toutes les clés sont uniques

Caractéristiques

- Recherche efficace : $O(log(n))$ mais peu dégénéré en $O(n)$
- Insertion et suppression rapides : Maintiennent l’ordre des éléments.
- Utilisation : Tables de recherche, ensembles dynamiques, files de priorité.
- Ordre des éléments : Permet une itération en ordre croissant.

Arbre AVL

Le concept d’arbre avl, se rapporte à des arbres de recherche qui mesure la hauteur de leurs sous arbre gauche et droite.

On modifie les fonction d’insertion et de supression pour rebalancer l’arbre.

- Insertion et supression plus lent.
- La recherche est garantie en $O(log(n))$.
- Un arbre binaire qui s’étale le plus possible
- Division des espaces de recherche

> https://sleek-think.ovh/index.php/cours/l3-algorithmie/38-abres-avl

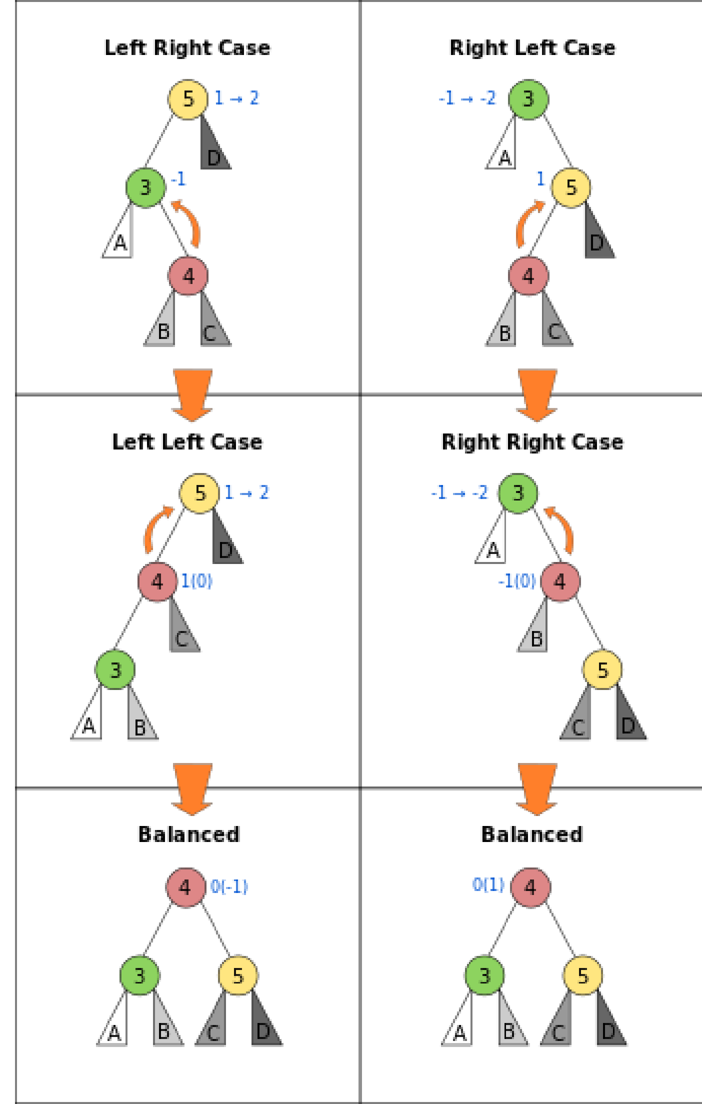


Table de hachage

- On utilise une fonction de hachage qui encode la clé de recherche
  - C'est une opération mathématique, idéalement en $O(1)$
  - La fonction de hachage doit éviter les collisions
  - S'il y a une collision, une resolution dois être proposé pour garantir l'unicité.
  - La fonction de hachage permet de `compresser` l'information facilitant la comparaison

```
unsigned int hash_function(char* key, int size) {
  unsigned int hash = 5381;
  int c;
  while ((c = *key++))
    hash = ((hash << 5) + hash) + c;
  return hash % size;
}
```

> On peu mixer les approches : arbre binaire + tableau de hachage

Graphe orienté et non-orienté

Il existe plusieurs types de représentation pour décrire un graphe.

Matrice d’adjacence

```
connection = np.zeros((n,n))
connection[0,0] = 1 # boucle sur x_1
connection[0,1] = 1 # arc de x_1 -> x_2
```

Matrice d’incidence sommet-arc

```
n = 2, m = 1 # number_of_nodes,  number_of_arcs
connection = np.zeros((n,m))
connection[0,0] = -1 # arc de x_1
connection[1,0] = 1 # vers x_2
```

Liste d’adjacence

```
nodes = [[] for _ in range(nomber_of_nodes)]
nodes[0] = [1] # liste des successeurs de x_1
nodes[1] = [0,2] # liste des successeurs de x_2
nodes[2] = [2,0] # boucle local
```

Traverser un graphe non-orienté

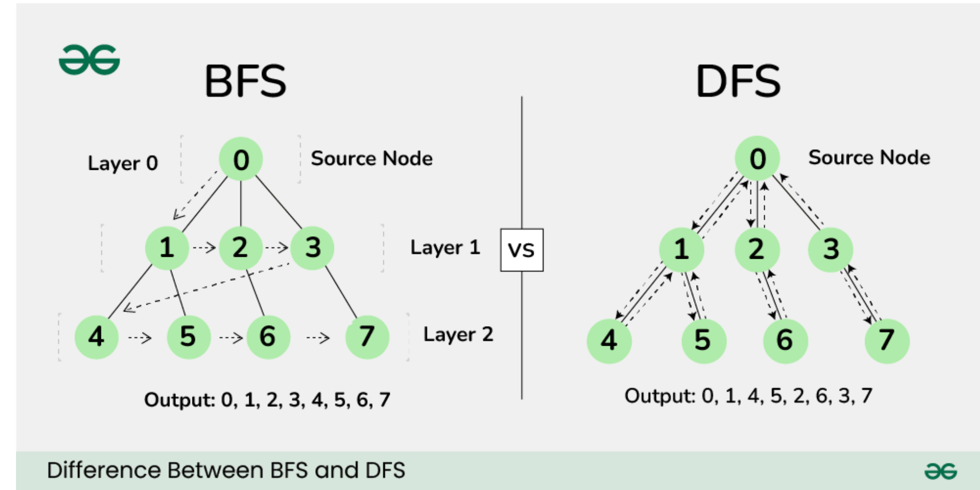
Recherche en largeur - Breadth First Search (BFS)

- FIFO (File d’attente) : Utilisée pour stocker les sommets à visiter.
- Initialisation : Ajouter le sommet de départ dans la file.
- Boucle :
  - Extraire le sommet de la file (FIFO).
  - Marquer le sommet comme visité.
  - Afficher le sommet.
  - Ajouter tous ses voisins non visités à la file.
- Répéter jusqu’à ce que la file soit vide.
- 5, 3, 10, 1, 2, 7, 20

https://sleek-think.ovh/index.php/cours/l3-maths-pour-l-info/32-theori-des-graphes
https://www.redblobgames.com/pathfinding/a-star/introduction.html
https://visualgo.net/en/dfsbfs?slide=1

Recherche en Profondeur - Depth First Search

- LIFO (Pile) : Sommets à visiter.
- Initialisation : sommet de départ sur la pile.
- Boucle :
  - Extraire le sommet du haut de la pile.
  - Marquer le sommet comme visité.
  - Afficher le sommet.
  - Add ses voisins non visités sur la pile.
- Répéter jusqu'à ce que la pile soit vide.
- 5, 3, 1, 2, 10, 7, 20



Depth First Search Algorithm | Graph Theory

Partager

Depth First Search
Algorithm

2 4 12 1 3 0 8 5 9 7 6 10 11

Regarder sur YouTube

TP FIFO - LIFO - Arbre binnaire - Graph

- (5 pts) Quelle structure de donnée est la plus adapté pour implémenter un LIFO et un FIFO ?
  - Comparer la complexité temporel et spatial des solutions, expliqué le choix (enqeue, dequeu, …)
  - Écrire un code de test de vos structures de donnée existant, permettant de faire une FIFO et un LIFO
- (7 pts) Implémenter les arbres abres binnaires de recherche `binary_search_tree.c`
  - Réécrire les fonctions de traverser d'arbres binnaire sans récursion
  - Insertion et suppression : garantir l'ordonancement
  - Utilise un pointeur sur fonction pour la traversé de l'arbre -> recherche/print/etc
- (8 pts) Implémenter un graph non-orienté `graph.c`
  - Utilisé une représentation par matrice d'adjacences
  - Implémenter BreadthFirstSearch et le DepthFirstSearch
  - Utilisé ces techniques pour compter le nombre de components
  - Vérifier vos algorithmes sur différents graphs
- Code degeu **-2** - pas d'explication / mini rapport **-2**

Niveau 4

Algorithme de trie et de recherche

Optimisation de recherche

En informatique, rechercher un élément dans un tableau est un problème récurant et parfois coûteux

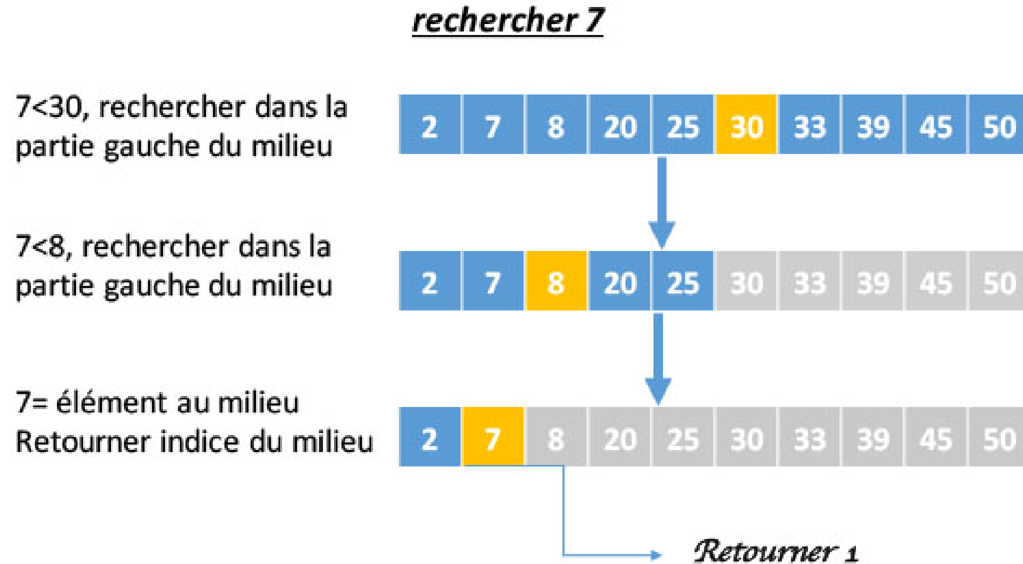
Lorsque les données ne sont pas triées / n'a pas de propriété sur les données

- Pas le choix, il faut vérifier tout les éléments

Lorsque les données sont triées

0. **Sémantique axiomatique** -> prouver formellement un programme
1. $\forall i, j \in [0, N], \; i < j \implies T[i] \leq T[j]$ (trié)
2. si $T[x] < \text{valeur}$
   - $\forall y \in [0..x], \; T[y] < \text{valeur} \wedge \exists w \in]x..N], \; T[w] = \text{valeur}$
3. si $T[x] > \text{valeur}$
   - $\forall y \in [x..N], \; T[y] > \text{valeur} \wedge \exists w \in [0..x[, \; T[w] = \text{valeur}$
4. En excluant une des deux parties à chaque itération, le nouvel ensemble a les mêmes propriétés
   - $\text{card}(U) < \text{card}(E)$ peut-être même $\text{card}(U) \approx \text{card}(E)/2$
   - **Divide and conquer**

Recherche dichotomique



- Implémenter une recherche dichotomique
- Quelle est la complexité temporelle de cet algorithme ?
- N puis N/2 puis N/4 etc

Trie Bulle

Le pire algorithme de trie existant

```
int i = 0, j = 0;
for (i = 0; i < n; i++) {   // loop n times
   for (j = 0; j < n-1; j++) { // what can be optimized here ?
      if (T[j] > T[j + 1])
          swap(T+j, T+j+1)
   }
}
```

- Quelle est la complexité spatial ?
- Quelle est la complexité temporel ? (pire et meilleur cas)
  - en terme de comparaison et de swap

Trie Boustrophedon

Le tri boustrophedon est une variante du tri à bulles

```
int start = 0, end = n - 1;

while (start < end) {
    for (int i = start; i < end; i++)
      // De gauche à droite, swap if arr[i] > arr[i + 1]
    end--;

    if(start == end)
        break;

    for (int i = end - 1; i >= start; i--)
        // De droite à gauche, swap if arr[i] > arr[i + 1])
    start++;
}
```

- Quelle est la complexité temporel ? (comparaison et swap)
- Que peut-on améliorer ?

Trie Boustrophedon

```
int k = 0;
while (2*k < n) {
    for (int i = k; i < n-k; i++)
        // Trouvé les indices min et max
    k++;

    // géré le cas min=max
    swap(T[min], ???);
    swap(T[max], ???)
}
```

- Quelle est la complexité temporel ? (comparaison et swap)
- Que peut-on améliorer ?
- oups : Selection sort

Trie par insertion

```
int i, key, j;
for (i = 1; i < n; i++) {
    key = T[i];
    j = i - 1;

    while (j >= 0 && T[j] > key) {
        T[j + 1] = T[j];
        j = j - 1;
    }

    T[j + 1] = key;
}
```

- Quelle est la complexité temporel ?

Trie Shell

Il s’agit d’une extension du tri par insertion, la différence essentielle étant que les éléments à comparer ne sont pas adjacents les uns aux autres, mais séparés par une certaine distance appelée "écart". L’écart est progressivement réduit jusqu’à ce qu’il devienne égal à 1. L’algorithme devient alors un tri par insertion.

```
static const int gaps[] = {
    // 1073, 281, 77, 23, 8, 1 // A036562 - 4/3
    // 1093, 364, 121, 40, 13, 4, 1 // A036562 - 3/2
    // 1750, 701, 301, 132, 57, 23, 10, 4, 1 // A102549 - 2001
    // 150, 85, 24, 9, 4, 1 // https://arxiv.org/pdf/2301.00316 - 2023 - 128
    // 488, 187, 72, 27, 10, 4, 1 // https://arxiv.org/pdf/2301.00316 - 2023 - 1000
};

for(int i = 0; i<sizeof(gaps)/sizeof(*gaps); ++i)
    shellsort_phase(array, length, gaps[i]); // insertion sort like
```

- Quelle est la complexité temporel ? (comparaison et swap)
- Quel intérét ?
- Cache-friendly ?

- (1pts) Implémenter un bubble sort
- (3pts) Implémenter la recherche dichotomique
  - (3pts) Bonnus : sans condition !
- (3pts) Implémenter un trie de boustrophedon amélioré
- (5pts) Implémenter un trie de shell
- (8pts) Implémenter un Heap Sort ou un Quick sort
- https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html