ALGORITHMIQUE ET STRUCTURES DE DONNÉES

4. Ensemble et Tableau associatif

Yoann Coudert--Osmont

29 Janvier 2026

1. Définitions

2. Arbres binaires de recherche

    Arbre AVL

    Arbre Rouge-Noir

3. Table de hachage

Définitions

Ensemble (Set)

Ensemble (Set)

Un Ensemble (ou Set) est un type abstrait stockant des valeurs non ordonnées et sans répétitions. Il s'agit d'une mise en œuvre de la notion mathématique d'ensemble fini. Les opérations possibles sont :

- **Ajout :** ajoute une nouvelle valeur à l'ensemble.
- **Recherche :** teste si une valeur appartient à l'ensemble.
- **Suppression :** supprime une valeur de l'ensemble.

Remarques

- Les valeurs sont **non ordonnées** : Si implémentation par tableau alors :
  | 1 | 4 | 2 | 5 | et | 4 | 5 | 2 | 1 | représentent le même Ensemble.

Remarques

- Les valeurs sont **non ordonnées** : Si implémentation par tableau alors :
  | 1 | 4 | 2 | 5 | et | 4 | 5 | 2 | 1 | représentent le même Ensemble.
- Les valeurs sont **sans répétitions** :
  | 4 | 5 | 2 | 1 | 5 | ne représente pas un ensemble.

Tableau associatif

Un tableau associatif est un type abstrait qui associe à un ensemble de **clefs**, un ensemble correspondants de **valeurs**. Un tableau associatif supporte quatre opérations :

- **Ajout :** associe une nouvelle valeur à une nouvelle clef.
- **Recherche :** retourne la valeur associée à une clef, si cette dernière existe.
- **Suppression :** supprime une clef du tableau ainsi que sa valeur associée.
- **Modification :** associe une nouvelle valeur à une clef déjà présente.

Implémentation avec un tableau (dynamique)

Implémentation	Ajout	Recherche	Suppression
Tableau	$\Theta(1)$ en moy. ($\Theta(n)$ pire cas)	$\Theta(n)$	$\Theta(1)$
Tableau trié	$\Theta(n)$	$\Theta(\log(n))$	$\Theta(n)$

Lorsque que le tableau est trié, on peut effectué une recherche dichotomique !

Suppression en temps constant

Lorsque le tableau n'est pas trié, la suppression est en $\Theta(1)$.
Pour un Ensemble E représenté par un `std::vector` en C++ on pourra supprimer la valeur se trouvant en position i comme suit :

```
void retirer(std::vector<int> &E, int i) {
    E[i] = E.back();
    E.pop_back();
}
```

Implémentation avec une liste chaînée

Implémentation	Ajout	Recherche	Suppression
Liste	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
Liste trié	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$

Lorsque que la liste est triée, on ne peut faire de recherches dichotomiques ! La complexité moyenne et pire cas est inchangée ($\Theta(n)$). En revanche la complexité dans le meilleur cas devient $\Theta(1)$ si l'élément recherché est inférieur ou égale au premier élément de la liste.

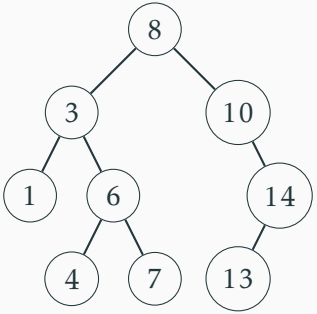
Question

Question : Peut-on faire mieux que des opérations en temps linéaire ?

Arbres binaires de recherche

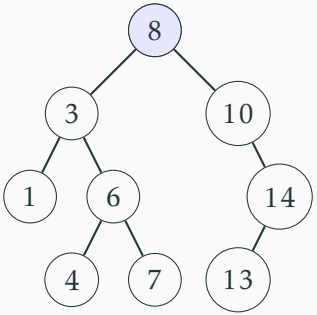
Arbre binaire de recherche

Un arbre binaire de recherche (ABR) est un arbre binaire dans lequel chaque nœud est supérieur aux nœuds dans son sous-arbre gauche et inférieur aux noeuds dans son sous-arbre droit.



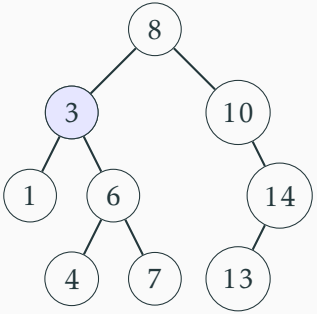
Opération recherche dans un ABR

On recherche la valeur 4 dans notre ABR :



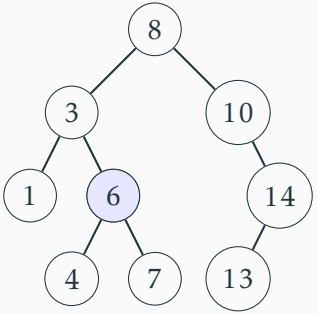
Opération recherche dans un ABR

On recherche la valeur 4 dans notre ABR :



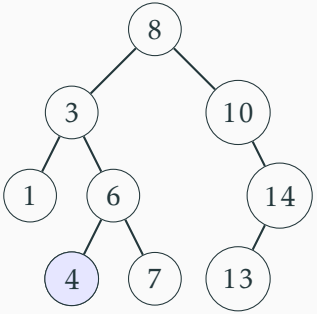
Opération recherche dans un ABR

On recherche la valeur 4 dans notre ABR :



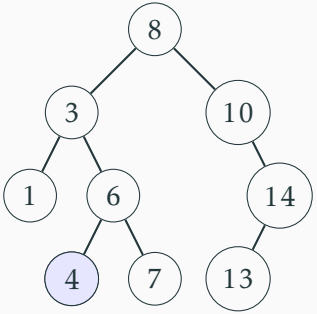
Opération recherche dans un ABR

On recherche la valeur 4 dans notre ABR :



Opération recherche dans un ABR

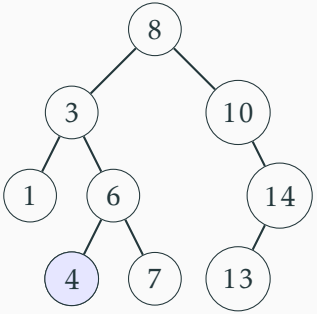
On recherche la valeur 4 dans notre ABR :



Le nombre de comparaison est au plus la hauteur H de l'arbre. La complexité est en $\Theta(H)$.

Opération recherche dans un ABR

On recherche la valeur 4 dans notre ABR :



Le nombre de comparaison est au plus la hauteur H de l'arbre. La complexité est en $\Theta(H)$.
L'arbre n'est pas un arbre binaire complet. Nous avons aucune garantie sur H hormis $H < n$. La complexité est donc en $\Theta(n)$.

Représentation d'un ABR

Un ABR se représente généralement par des cellules qui pointent entre elles à la manière d'une liste chaînée.

```
struct Cellule {
    int valeur;
    Cellule *pere, *fild_droit, *fils_gauche;
};

using ABR = Cellule*; // pointeur vers la racine
```

Implémentation de la recherche en C++

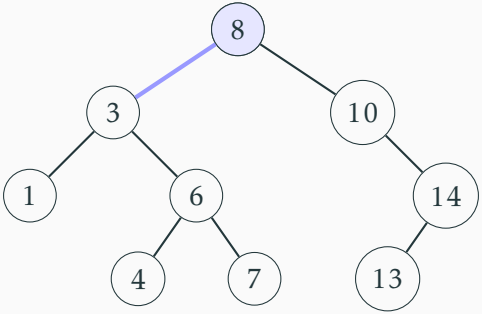
Voici une implémentation récursive de la recherche de la valeur x dans un ABR A :

```
Cellule* recherche(ABR A, int x) {
    if(A == nullptr) return nullptr;
    if(x == A.valeur) return A;
    if(x < A.valeur) return recherche(A->fils_gauche, x);
    else return recherche(A->fils_droit, x);
}
```

Lorsque que la valeur x n'est pas trouvée, la fonction retourne `nullptr`.

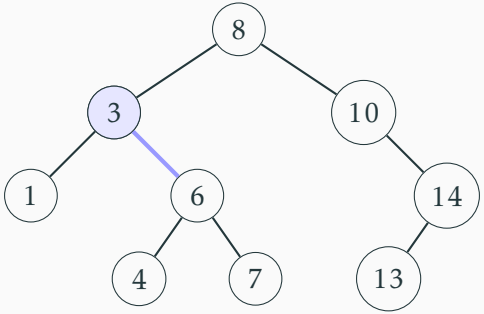
Opération ajouter dans un ABR

On ajoute la valeur 5 dans notre ABR :



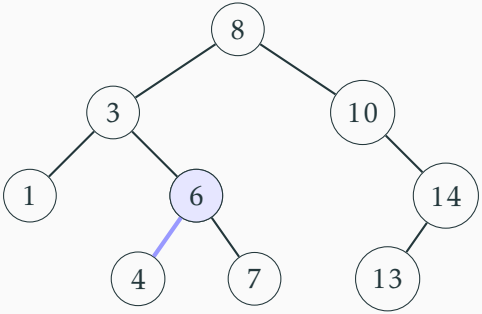
Opération ajouter dans un ABR

On ajoute la valeur 5 dans notre ABR :



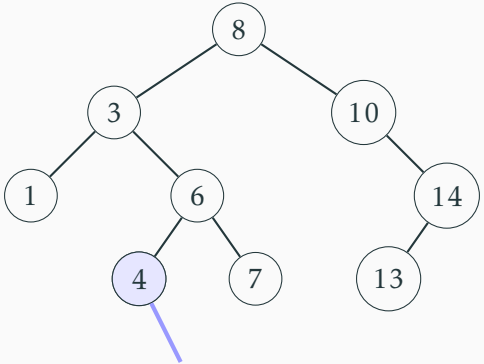
Opération ajouter dans un ABR

On ajoute la valeur 5 dans notre ABR :



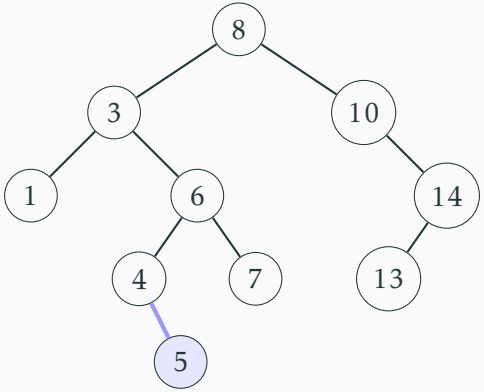
Opération ajouter dans un ABR

On ajoute la valeur 5 dans notre ABR :



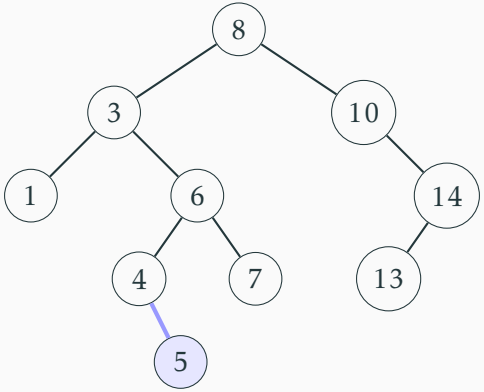
Opération ajouter dans un ABR

On ajoute la valeur 5 dans notre ABR :



Opération ajouter dans un ABR

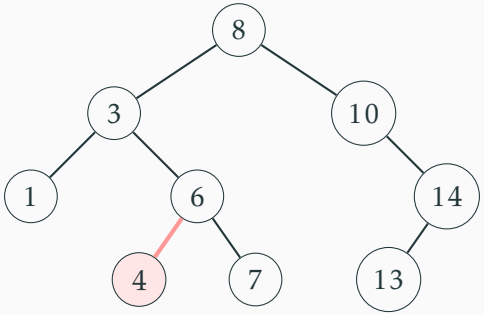
On ajoute la valeur 5 dans notre ABR :



La complexité pire cas est en $\Theta(H) = \Theta(n)$.

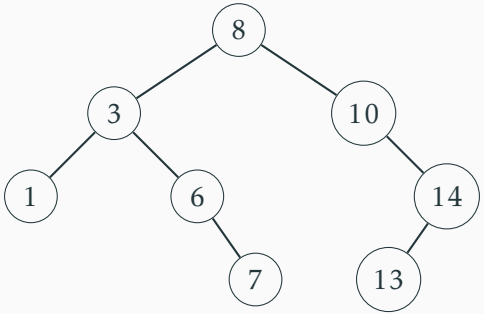
Opération suppression dans un ABR

On supprime la valeur 4 dans notre ABR :



Opération suppression dans un ABR

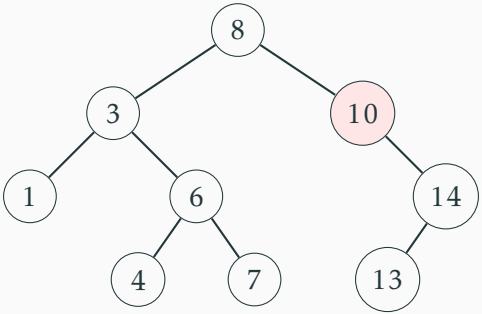
On supprime la valeur 4 dans notre ABR :



La complexité est en $\Theta(1)$ lorsqu'on supprime une feuille.

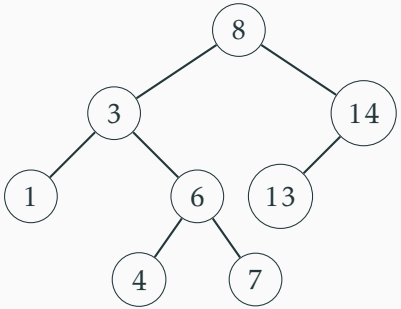
Opération suppression dans un ABR

On supprime la valeur 10 dans notre ABR :



Opération suppression dans un ABR

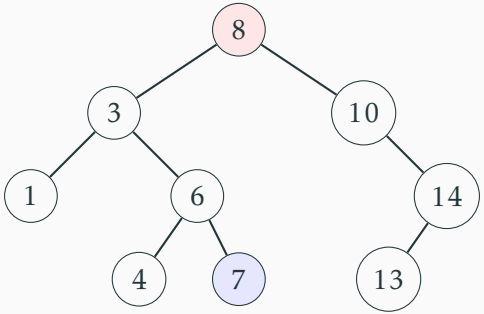
On supprime la valeur 10 dans notre ABR :



La complexité est en $\Theta(1)$ lorsqu'on supprime un nœud possédant un seul fils.

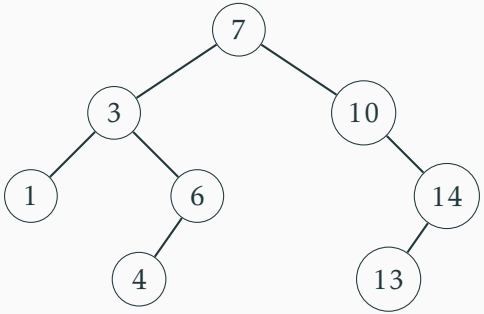
Opération suppression dans un ABR

On supprime la valeur 8 dans notre ABR :



Opération suppression dans un ABR

On supprime la valeur 8 dans notre ABR :



Si A est le sous-arbre de l'élément à supprimer, et G le sous-arbre gauche de A. Alors on trouve le maximum x de G, on le supprime et on remplace la racine de A par x.
La complexité est en $\Theta(H) = \Theta(n)$.

Implémentation de la suppression en C++

Voici une implémentation récursive de la suppression de la racine d'un ABR :

```
ABR suppression(ABR A) {
    ABR B = nullptr;
    if(A->fils_gauche) {
        if(A->fils_droit) {
            B = max(A->fils_gauche);
            A->valeur = B->valeur;
            suppression(B);
            return A;
        } else B = A->fils_gauche;
    } else if(A->fils_droit) B = A->fils_droit;
    if(A->pere) {
        if(A->pere->fils_gauche == A) A->pere->fils_gauche = B;
        if(A->pere->fils_droit  == A) A->pere->fils_droit  = B;
    }
    if(B) B->pere = A->pere;
    delete A;
    return B;
}
```

Récapitulatif

Pire cas :

Implémentation	Ajout	Recherche	Suppression
Tableau	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
Tableau trié	$\Theta(n)$	$\Theta(\log(n))$	$\Theta(n)$
Liste	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
Liste trié	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
ABR	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

Récapitulatif

Pire cas :

Implémentation	Ajout	Recherche	Suppression
Tableau	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
Tableau trié	$\Theta(n)$	$\Theta(\log(n))$	$\Theta(n)$
Liste	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
Liste trié	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
ABR	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

On peut montrer qu'en moyenne $H = \Theta(\log(n))$.

Cas Moyen :

Implémentation	Ajout	Recherche	Suppression
Tableau	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
Tableau trié	$\Theta(n)$	$\Theta(\log(n))$	$\Theta(n)$
Liste	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
Liste trié	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
ABR	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$

Question

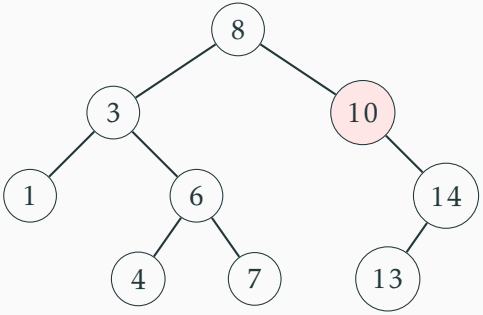
Question : Peut-on également obtenir du $\Theta(\log(n))$ dans le pire cas ?

Arbres binaires de recherche

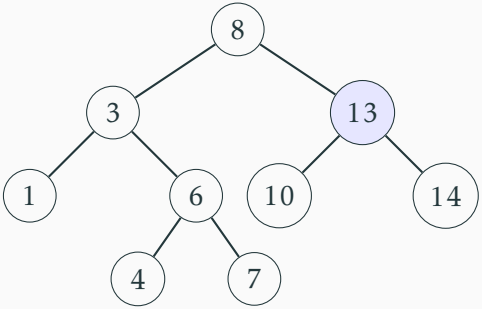
Arbre AVL

Arbre AVL

Un arbre AVL est un ABR tel que pour chaque nœud la différence de hauteur entre son fils droit et son fils gauche est au plus 1.



ABR non AVL



ABR AVL

Analyse de la hauteur

On pose $N_{\min}(h)$ le nombre minimum de nœuds dans un arbre AVL de hauteur h.

$$N_{\min}(0) = 1 \qquad N_{\min}(1) = 2$$

Un arbre AVL contient moins de sommets lorsqu'il est déséquilibré. Donc :

$$N_{\min}(h+1) = 1 + N_{\min}(h) + N_{\min}(h-1)$$

Asymptotiquement $N_{\min}$ se comporte comme la suite de Fibonacci :

$$N_{\min}(h) = \Theta\left(\varphi^h\right)$$

En passant au $\log_2$ on obtient : $\log_2(N_{\min}(h)) \sim h \cdot \log_2 \varphi$. Ainsi la hauteur maximum $H_{\max}(n)$ d'un arbre AVL de taille n est :

$$H_{\max}(n) \sim \frac{\log_2(n)}{\log_2 \varphi} \simeq 1,44 \cdot \log_2(n)$$

Complexité

Pire cas :

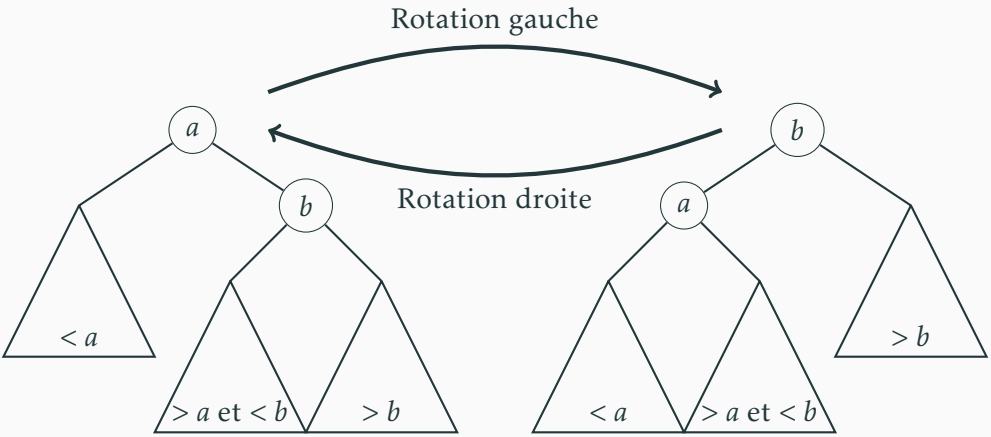
Implémentation	Ajout	Recherche	Suppression
Tableau	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
Tableau trié	$\Theta(n)$	$\Theta(\log(n))$	$\Theta(n)$
Liste	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
Liste trié	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
ABR	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Arbre AVL	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$

Opérations sur un arbre AVL

Les opérations sur un arbre AVL sont les mêmes que sur un ABR, mais après avoir modifié un nœud de l'arbre, il faut le rééquilibré pour que les hauteurs de ses sous-arbres droit et gauche diffèrent d'au plus 1.
Pour cela il faut utiliser une nouvelle opération appelé rotation.

Opération de rotation

Une rotation consiste à échanger la racine d'un arbre avec l'un de ses fils.



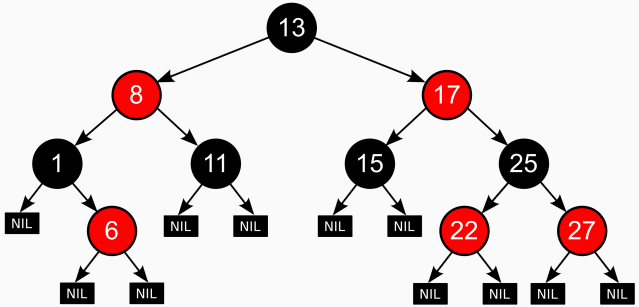
Arbres binaires de recherche

Arbre Rouge-Noir

Arbre Rouge-Noir

Un arbre rouge noir est un ABR vérifiant les propriétés suivantes :

- Chaque nœud est soit rouge soit noir.
- Les enfants d'un nœud rouge sont noirs.
- Les feuilles sont noires et n'ont pas de valeurs.
- Tous les chemins descendant d'un même nœud vers les feuilles ont le même nombre de nœuds noirs.



Analyse de la hauteur

Soit un arbre Rouge-Noir de taille n et de hauteur H. On pose n_n, le nombre de nœuds noirs par branche depuis la racine.
D'après la dernière propriété, les n_n premiers niveaux de l'arbre sont complets :

$$n \geqslant 2^{n_n} - 1$$

La deuxième propriété implique que chaque broche a au plus n_n nœuds rouges :

$$H < 2 \cdot n_n$$

Donc $n \geqslant 2^{H/2}$. Ce qui veut dire, en passant au $\log_2$:

$$H \leqslant 2 \cdot \log_2(n)$$

Complexité

Pire cas :

Implémentation	Ajout	Recherche	Suppression
Tableau	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
Tableau trié	$\Theta(n)$	$\Theta(\log(n))$	$\Theta(n)$
Liste	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
Liste trié	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
ABR	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Arbre AVL	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$
Arbre Rouge-Noir	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$

Dans les langages de programmation

La librairie standard de C++ contient des arbres binaires de recherche implémentant des Ensembles.

```cpp
#include <set>
std::set<int> S; // Un Ensemble d'entiers
S.insert(x); // Ajoute x à l'Ensemble
S.find(x); // Cherche x dans l'Ensemble
S.erase(x); // Supprime la valeur x de l'ensemble
```

De la même manière, on trouve des tableau associatifs.

```cpp
#include <map>
// Un tableau associatif dont les clefs sont des entiers
// et les valeurs des strings
std::map<int, string> M;
M[x] = s; // Modifie la valeur de la clef x
```

Table de hachage

Fonction de hachage

Une fonction de hachage est une fonction h qui prend en entrée une donnée stockée sur un nombre arbitraire de bits et retourne un entier tenant sur un nombre b fixé de bits.

Une fonction de hachage doit **avoir l'air aléatoire**. Si x et y sont proches, alors $h(x)$ et $h(y)$ ne doivent pas être proches.

Elle doit être **difficile à inverser**. Si $0 \leqslant y < 2^b$, alors il doit être difficile de trouver x tel que $h(x) = y$.

Exemples

MD5 et SHA256 sont des fonctions de hachages de tailles 128 et 256 bits.

```
>>> import hashlib

>>> hashlib.md5(b"coucou").hexdigest()
'721a9b52bfceacc503c056e3b9b93cfa'
>>> len(hashlib.md5(b"coucou").hexdigest())*4
128

>>> hashlib.sha256(b"coucou").hexdigest()
'110812f67fa1e1f0117f6f3d70241c1a42a7b07711a93c2477cc516d9042f9db'
>>> hashlib.sha256(b"nounou").hexdigest()
'8ed8dc9a684052fc5d9a61464087059ca32044d8342116854b78a81579196727'
```

Usage

Les fonctions de hachages sont utiles dans de nombreux contextes :

- Stocker les mots de passe

Usage

Les fonctions de hachages sont utiles dans de nombreux contextes :

- Stocker les mots de passe
- Vérifier si des données envoyées ont été correctement reçu.

Usage

Les fonctions de hachages sont utiles dans de nombreux contextes :

- Stocker les mots de passe
- Vérifier si des données envoyées ont été correctement reçu.
- La sécurité des blockchains reposent sur les fonctions de hachage.

Usage

Les fonctions de hachages sont utiles dans de nombreux contextes :

- Stocker les mots de passe
- Vérifier si des données envoyées ont été correctement reçu.
- La sécurité des blockchains reposent sur les fonctions de hachage.
- Et bien sûr les tables de hachage...

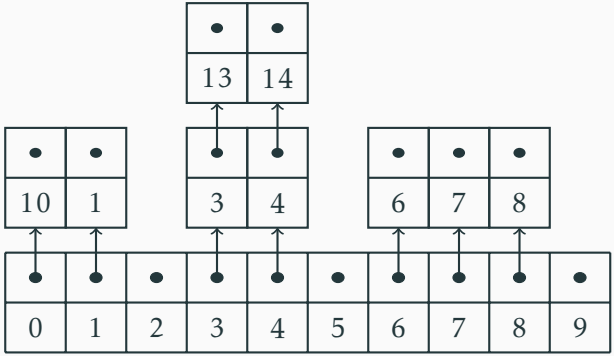
Table de hachage

Une table de hachage est une structure de données implémentant un Ensemble à l'aide d'une fonction de hachage h.

Les valeurs sont stockés dans une table (un tableau). Une valeur x sera stocké à l'indice $h(x)$ dans la table. Les valeurs stockés à un même indice sont organisés sous forme d'une liste chaînée.

Ensemble
$\{1, 3, 4, 6, 7, 8, 10, 13, 14\}$

$h(x) = [x \mod 10]$



Analyse

On note L_i, la liste à l'indice i de la table.

- **Recherche :** Pour chercher si x est présent, il faut parcourir la liste $L_{h(x)}$. La complexité est donc $\Theta\left(\left|L_{h(x)}\right|\right)$.
- **Ajout :** Lors de l'ajout d'une valeur x, il faut vérifier que x n'est pas présent dans $L_{h(x)}$. La complexité est alors $\Theta\left(\left|L_{h(x)}\right|\right)$.
- **Suppression :** La suppression correspond à la suppression d'un élément dans une liste. En la supposant doublement chaînée, la complexité est $\Theta(1)$.

Analyse

On note L_i, la liste à l'indice i de la table.

- **Recherche :** Pour chercher si x est présent, il faut parcourir la liste $L_{h(x)}$. La complexité est donc $\Theta\left(\left|L_{h(x)}\right|\right)$.
- **Ajout :** Lors de l'ajout d'une valeur x, il faut vérifier que x n'est pas présent dans $L_{h(x)}$. La complexité est alors $\Theta\left(\left|L_{h(x)}\right|\right)$.
- **Suppression :** La suppression correspond à la suppression d'un élément dans une liste. En la supposant doublement chaînée, la complexité est $\Theta(1)$.

Bien que dans le pire cas $\left|L_{h(x)}\right| = n$, en moyenne, si la fonction de hachage est bonne, $\left|L_{h(x)}\right| = \Theta(1)$.

Complexité

Pire cas :

Implémentation	Ajout	Recherche	Suppression
Tableau	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
Tableau trié	$\Theta(n)$	$\Theta(\log(n))$	$\Theta(n)$
Liste	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
Liste trié	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
ABR	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Arbre AVL	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$
Arbre Rouge-Noir	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$
Table de hachage	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$

Complexité

Cas Moyen :

Implémentation	Ajout	Recherche	Suppression
Tableau	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
Tableau trié	$\Theta(n)$	$\Theta(\log(n))$	$\Theta(n)$
Liste	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
Liste trié	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
ABR	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$
Arbre AVL	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$
Arbre Rouge-Noir	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$
Table de hachage	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

Dans les langages de programmation

La librairie standard de C++ contient des tables de hachage implémentant des Ensembles.

```
#include <unordered_set>
std::unordered_set<int> S; // Un Ensemble d'entiers
S.insert(x); // Ajoute x à l'Ensemble
S.find(x); // Cherche x dans l'Ensemble
S.erase(x); // Supprime la valeur x de l'ensemble
```

De la même manière, on trouve des tableau associatifs.

```
#include <unordered_map>
// Un tableau associatif dont les clefs sont des entiers
// et les valeurs des strings
std::unordered_map<int, string> M;
M[x] = s; // Modifie la valeur de la clef x
```

Dans les langages de programmation

Python possède également des tables de hachages implémentant des Ensembles.

```
>>> set([4, 5, 4, 9])
{4, 5, 9}
>>> S = {3, 8, 56}
>>> S.add(x) # Ajoute x
>>> S.remove(x) # Supprime x
>>> x in S # Test si x est dans S
```

De la même manière, on trouve des tableau associatifs.

```
>>> dict([(4, "coucou"), (5, "salut")])
{4: 'coucou', 5: 'salut'}
>>> d = {"a": 42, 12: "test"}
>>> d[x] = y # Modifie la valeur de clef x
```