Introduction

Algorithme : composition d'un ensemble fini d'étapes formées d'un ensemble fini d'opérations dont chacune est définie de façon rigoureuse et non ambiguë.

Notion de complexité

L'efficacité d'un algorithme se mesure en fonction d'un paramètre n qui caractérise la quantité d'éléments traités par l'algorithme.

La complexité caractérise l'efficacité d'un algorithme et fournit une estimation de son coût en temps processeur en fonction de la taille de données.

La complexité est calculée approximativement dans le sens où l'on donnera simplement un ordre de grandeur par rapport au nombre d'opérations élémentaires nécessaires pour effectuer un algorithme

La notation de Landau « O » (grand O)

Définition : La fonction f est un grand O de la fonction g, (« f croît au plus aussi vite que g ») et l'on note f = O(g) ou f(n)=O(g(n)) s'il existe une constante réelle positive c et un entier positif n0 tels que f(n) <= c*g(n), pour tout n >= n0. On dit que g(n) est une borne supérieure asymptotique pour f(n)

log(n) < n < n*log(n) < n^2 < n^3 < 2^n

Définitions « o »

Pour des fonctions f et g nous écrivons f=o(g) si lim(n->inf) (f(n)/g(n) = 0

Introduction : type de données abstrait (TDA)

Motivation :

> La notion de TDA existe déjà dans le cas des types simples
>
> En déclarant une variable d'un type simple (par exemple réelle), on obtient implicitement des opérations (+, -, *, /)

Notion de type de données abstrait

> Un type de données abstrait (TDA) est défini par un type de données et ensemble des opérations permettant de gérer ces données, les détails de l'implémentation restant cachés.

Introduction : paradigme du TDA

Abstraction des données :

> Consiste à identifier les caractéristiques importantes de l'entité vis-à-vis des applications envisagées.
>
> L'abstraction permet la description simple et précise du TDA

Encapsulation des données

> Consiste à intégrer dans la même entité une structure de données et les opérations chargées de la gérer.

L'encapsulation permet de masquer les détails de l'implémentation

Modularité

Est la faculté de diviser un gros système en unités plus petites.

La modularité favorise la réutilisation du code. Les modules groupés par fonctionnalités générales peuvent être intégrés dans d'autres applications

Récursivité

Définition

Notion de récursivité :

Une entité est définie récursivement si la définition comporte la mention de l'entité

Un processus de traitement est récursif s'il se réfère à lui-même

Mise en œuvre

Cas général

Décomposer le problème en sous-problèmes ; la solution du problème s'exprimera alors en fonction des solutions es sous-problème

Paramètres de la récursivité

Déterminer quelles sont les données qui interviennent dans l'algorithme, en particulier celles qui distinguent les différents niveaux de récursivité

Cas trivial

Trouver une ou plusieurs conditions de terminaison de l'algorithme (l'algorithme donne la solution sans appel récursif), conditions écrites en fonction des paramètres de la récursivité.

Algorithme des tours de Hanoï

Transférer n disques d'un piquet d'origine sur un piquet destination

Si le nombre n de disques à transférer > 0 alors
Transférer n-1 disques du piquet d'origine sur le troisième piquet en utilisant le piquet destination comme intermédiaire
Transférer le disque restant du piquet d'origine sur le piquet destination
Transférer n-1 disques du troisième piquet sur le piquet destination en utilisant le piquet d'origine comme intermédiaire
Fin si

```
void deplacertour(unsigned int n, piquet_t p1, piquet_t p2, piquet_t p3){
    if (n > 0) {
        /* tour(n) = un gros disque D et une tour(n - 1) */
        deplacertour(n - 1, p1, p3, p2); /* tour(n - 1): p1 -> p2 */
        deplacerdisque(p1, p3); /* D: 1 -> 3 */
        deplacertour(n - 1, p2, p1, p3); /* tour(n - 1): p2 -> p3 */
    }
}
```

Les courbes de Hilbert

Dessiner Hn (orientation)

- Pivoter de 90 degrés dans la même orientation
- Dessiner H_{n-1}(autre orientation)
- Dessiner un trait
- Pivoter de 90 degrés dans l'autre orientation
- Dessiner H_{n-1} (orientation)
- Dessiner un trait
- Dessiner H_{n-1}(orientation)
- Pivoter de 90 degrés dans l'autre orientation
- Dessiner un trait
- Dessiner H_{n-1}(autre orientation)
- Pivoter de 90 degrés dans la (même) orientation

La complexité de l'algorithme de dessin des courbes de Hilbert peut être caractérisée par le nombre n de traits à dessiner.

Pour dessiner la courbe H_1 il faut 3 traits : n = 3

Pour dessiner la courbe H_2 il faut 15 (4*3 + 3) traits

Pour dessiner la courbe H_3 il faut 63 (4*15+3) traits

La formule générale $4^n - 1$ pour H_n peut être facilement déduite $O(4^n)$

Tris

Tris bulles

```
Pour i variant de 1 à n-1 boucler
    Pour j variant de 1 à n-i boucler
        Si tableau(j) > tableau(j+1) alors
            Permuter les éléments tableau(j) et tableau(j+1)
        Fin si
    Fin boucler
Fin boucler
```

Tris par insertion

Pour i variant de 2 à n boucler

Affecter la valeur de tableau(i) à la variable tampon

Affecter la valeur de i à la variable j

Tant que j-1 >= 1 et tableau(j-1) > tampon boucler

Affecter la valeur de tableau(j-1) à la variable tableau(j)

Décrémenter j de 1

Fin boucler

Affecter la valeur de tampon à la variable tableau(j)

Fin boucler

Tris par sélection

Pour i variant de 1 à n-1 boucler

Affecter la valeur de i à la variable imin

Pour j variant de i+1 à n boucler n

Si tableau(j) < tableau(imin) alors

Affecter la valeur de j à la variable imin

Fin si

Fin boucler

Permuter tableau(i) et tableau(imin)

Fin boucler

Tri rapide

Est dans le pire des cas en $O(n^2)$

Est l'un des tris les plus intéressants à cause de son efficacité dans le cas moyen qui est de O(n*log(n))

Le tri rapide est basé sur un processus de partitionnement

Appliqué récursivement

Qui subdivise le tableau à trier en deux sous-tableaux définis ainsi :

Le sous-tableau de gauche ne comporte que des éléments inférieurs ou égaux à une valeur de référence qui fait partie des valeurs du tableau

Le sous-tableau de droite ne comporte que des éléments supérieurs ou égaux à la valeur de référence

```
Si dernier > premier alors  (cas trivial, un seul élément)
    Affecter la valeur tableau(premier+dernier/2) à la variable reference
    Affecter la valeur de premier à la variable gauche
    Affecter la valeur de dernier à la variable droite
    Tant que gauche <= droite boucler  (gauche et droite pas croisés)
        Tant que tableau(gauche) < reference boucler
            Incrémenter gauche de 1
        Fin boucler
        Tant que tableau(droite) > reference boucler
            Décrémenter droite de 1
        Fin boucler
        Si gauche < droite alors (gauche et droite pas croisé)
            Permuter tableau(gauche) et tableau(droite)
            Incrémenter gauche de 1
            Décrémenter droite de 1
        Sinonsi gauche = droite alors  (forcer le partage)
            Si gauche < dernier alors
                Incrémenter gauche de 1
            Fin si
            Si droite > premier alors
                Décrémenter droite de 1
            Fin si
        Fin si
    Fin boucler (tant que gauche <= droite, c-à-d pas croisés)
    Tri rapide du tableau bornes premier et droite
    Tri rapide du tableau de borne gauche et dernier
Fin si
```

Tris par tas

(première étape : réaliser un tas)

Tant que j <= n/2 boucler (l'élément a encore au moins un fils)

(trouver le plus grand fils de l'élément)

Affecter la valeur 2*j à la variable position

Si position < n alors (l'élément à deux fils)

Si tableau(position+1) > tableau(position) alors

Affecter la valeur position+1 à la variable position

Fin si

Fin si

(position désigne le plus grand fils de l'élément)

Terminer si tableau(j) >= tableau(position)

(permuter l'élément avec son plus grand fils)

Permuter tableau(j) et tableau(position)

Affecter la valeur de position à la variable j

Fin boucler

(deuxième étape : trier)

Affecter la valeur n à la variable taille

Boucler

Permuter tableau(1) et tableau(taille)

Décrémenter de 1 la variable taille

Descendre l'élément de position 1 dans le tableau partiel

Terminer si taille <= 1

Fin boucler

Complexité en O(n*log(n))

Analyse de la complexité

Cas du tri bulles

T(n) = (n-1)*b1 + $\sum_{i=1}^{n-1} Tint(i)$

Tint(i) = n-1*b2 + (n-i) * b3 + (n-i) * b4 = (n-i) * (b2 + b3 + b4)

T(n) = (n-1) * b1 + (n(n-1)/2) * (b2 + b3 + b4) =

n^2 (b2 + b3 + b4)/2 + n(b1 – (1/2) *(b1 + b2 + b3) – b4

≤ n^2 (b2 + b3 + b4) /2 + n^2 * b1 + (n^2/2) (b2 + b3 + b4) + b1 * n^2

≤ c * n^2

Structures linéaires

Motivations

Les structures linéaires sont des fichiers, des listes, des piles, des queues, …

Les éléments d'une structure linéaire sont placés dans un certain ordre, ce qui implique qu'un même parcours fournira toujours les éléments dans le même ordre.

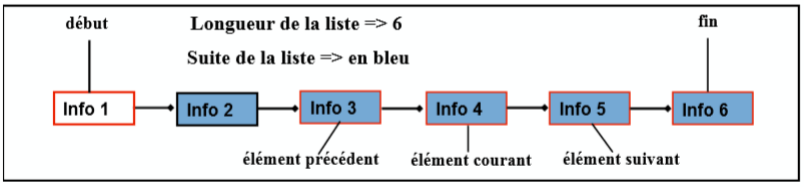
Définition

Une liste (ou fichier) est formée d'éléments successifs, classés ou non, dont le nombre peut être bornée ou illimité.

Exemple :

Une file d'attente devant un guichet ou la liste de passage à un examen.

Une liste sert principalement à représenter une collection d'informations qui peuvent ou doivent se suivre.



On appelle début ou tête le premier, respectivement fin le dernier élément d'une liste. La suite d'une liste est la liste formée de tous les éléments sauf celui du début. Comme les éléments se suivent, l'élément suivant et l'élément précédent d'un élément donné sont définis (sauf pour ceux au début et à la fin). L'élément considéré (traité) à un moment donné est appelé élément courant. Le nombre d'éléments s'appelle longueur de la liste.

Opérations sur les listes

Les opérations classiques sur les listes sont :

- Insérer une information, la rajouter à la liste
- Supprimer ou extraire une information, l'éliminer de la liste
- Modifier un élément, changer l'information présente

- Consulter un élément, c-à-d lire l'information présente dans l'élément
- Savoir si la liste est vide
- Rechercher une information particulière, c-à-d savoir si elle fait partie de la liste ou non
- Parcourir la liste, c-à-d effectuer un traitement sur chacun de
- ses éléments

Ces opérations sont les plus simples et forment la base du traitement des listes. L'endroit où un élément sera rajouté ou supprimé joue un rôle important dans la manipulation de certaines listes.

Queue

L'insertion se passe à la fin et la suppression au début ce qui conduit au fait que le premier élément mis dans la queue en sera le premier extrait. FIFO (first in first out)

Ça correspond à la situation d'attente devant un guichet, la personne située au début de la queue la quitte après avoir dialogué avec l'employé au guichet et les personnes qui arrivent se placent à la fin de la queue

Pile

Si l'insertion et la suppression ont lieu toutes les deux au début ce qui conduit au fait que le premier élément empilé sera le dernier désempilé. LIFO (last in first out)

Une pile se manipule comme une pile d'assettes qui sont prises et remises toujours au début, tout en haut des assiettes restantes.

Queue de priorité

Si l'insertion s'effectue selon un critère d'ordre et la suppression au début.

La gestion d'une file d'impressions sur une imprimante s'effectue très souvent selon une queue de priorité où les fichiers à imprimer sont classés par ordre croissant de taille. Le fichier imprimé puis éliminé est toujours pris au début (c'est le plus petit) alors qu'un nouveau fichier est placé dans la file en fonction de sa taille.

Arbre

Motivation

Un arbre généraliser une liste (1D à 2D) et permet d'avoir une insertion, une suppression et une recherche à O(log(n)).

Les applications classiques d'un arbre sont :

- réalisation d'algorithmes de tri de complexité O(n log(n))
- réalisation d'algorithmes de recherche de complexité O(log(n))
- implémentation de queues de priorités
- implémentation de la table des symboles d'un compilateur
- conversion en notation postfixée d'expressions arithmétiques écrites en notation infixée
- résolution de problèmes d'indexation

Vue d'ensemble

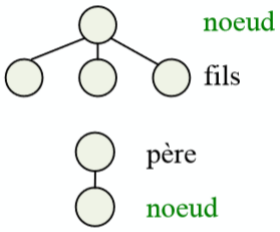
Les arbres sont catégorisés de la façon suivante :

- arbres arbitraires
- arbres binaires
  - arbres binaires (sans contrainte particulière)
  - arbres binaires organisés en tas
  - arbres (binaires) de recherche
  - arbres (binaires) de recherche équilibrés

Définitions associées à un arbre

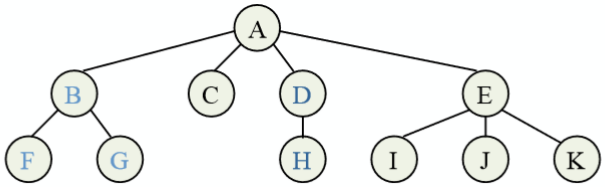
Un arbre est une collection de nœuds répartis en 2D et associés selon un ordre hiérarchique strict

Un nœud a



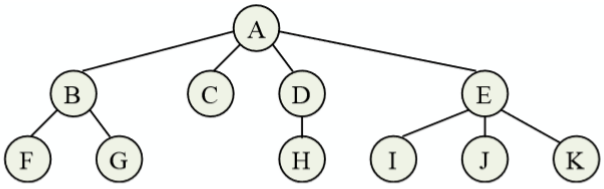
- 0-n nœuds descendants (fils)
- 1 nœud ascendant (père)

Une racine est le premier nœud de la hiérarchie et le seul nœud n'ayant pas de père.



Arbre vide	empty tree		arbre sans nœuds
Sous-arbre	subtree	(F,B,G) ou (D,H)	sous-ensemble de nœuds d'un arbre ayant une structure d'arbre
Degré d'un arbre	tree degree		degré maximum de ses nœuds
Hauteur	height	elle vaut 3	nombre de nœuds du chemin le plus long

Définition associées à un nœud

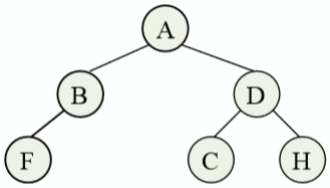


Terme	Terme anglais	Exemples	Définition
Feuille	leaf	F, G, C, H, I, J, K	nœud sans fils
Nœud interne	internal node		nœuds avec fils
Etiquette	label	A	information associée à un nœud
Degré d'un nœud	node degree		nombre de ses fils
Chemin d'un nœud	path	(A, D, H)	suite de nœuds reliants la racine à un nœud
Niveau d'un nœud	level		nombre de nœuds sur le chemin d'un nœud

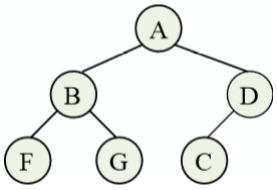
Cas particulier

Un arbre plein est un arbre tel que en tout nœud N, à l'exception (éventuelle) de l'avant-dernier niveau :

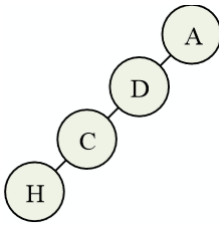
degré(N) = degré(arbre)



Un arbre complet est un arbre avec un remplissage du dernier niveau depuis la gauche.

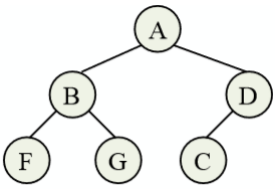


Un arbre dégénéré est un arbre tel que le degré de chaque nœud interne vaut 1.



Définition d'un arbre binaire

Un arbre binaire est un arbre de degré 2.



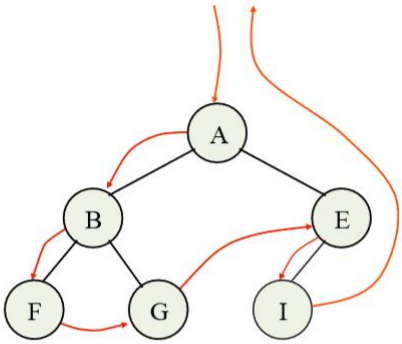
Le fils de gauche est le premier descendant et le fils de droite est le second descendant.

L'intérêt d'un arbre binaire est qu'il le plus simple, en général aussi performant que des structures plus complexes et à la base de majorité des autres arbres.

Parcours

Préordonné

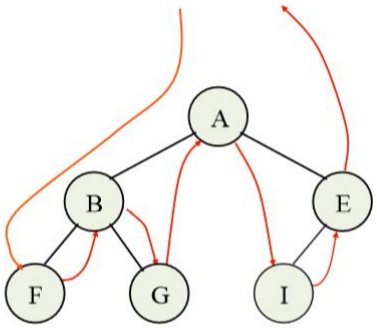
```
si l'arbre n'est pas vide alors
   traiter la racine de l'arbre
   effectuer le parcours préordonné de son sous-arbre gauche
   effectuer le parcours préordonné de son sous-arbre droit
fin si
```



- nœuds parcourus : A , B , F , G , E , I

Symétrique

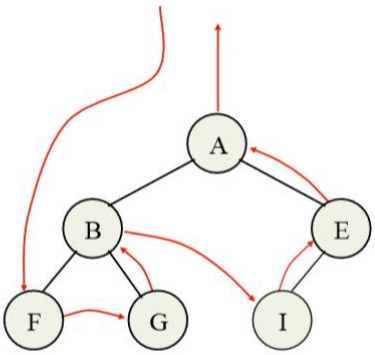
```
si l'arbre n'est pas vide alors
   effectuer le parcours symétrique de son sous-arbre gauche
   traiter la racine de l'arbre
   effectuer le parcours symétrique de son sous-arbre droit
fin si
```



- nœuds parcourus : F , B , G , A , I , E

Postordonné

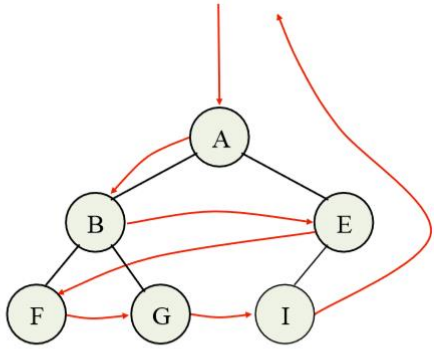
```
si l'arbre n'est pas vide alors
   effectuer le parcours postordonné de son sous-arbre gauche
   effectuer le parcours postordonné de son sous-arbre droit
   traiter la racine de l'arbre
fin si
```



- nœuds parcourus : F , G , B , I , E , A

En largeur

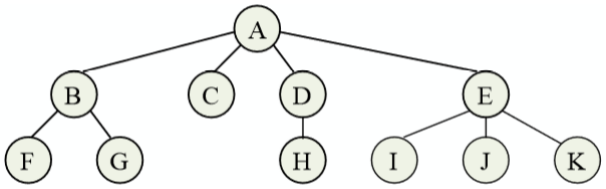
```
si l'arbre n'est pas vide alors
    déposer la racine de l'arbre dans la queue
    tant que la queue n'est pas vide boucler
       prélever un nœud dans la queue
       traiter ce nœud
       pour chaque fils de ce nœud, pris de gauche à droite boucler
            déposer ce nœud dans la queue
       fin boucler
   fin boucler
fin si
```



- nœuds parcourus : A , B , E , F , G , I

Définition d'un arbre arbitraire

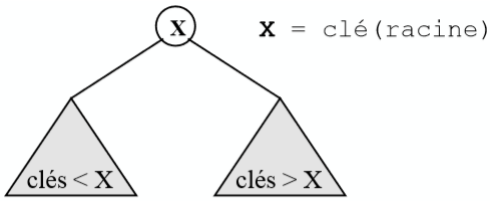
Un arbre arbitraire est un arbre n-aire ou un arbre de degré n. Les fils d'un nœud peuvent être numérotés de 1 à n pour être identifié.

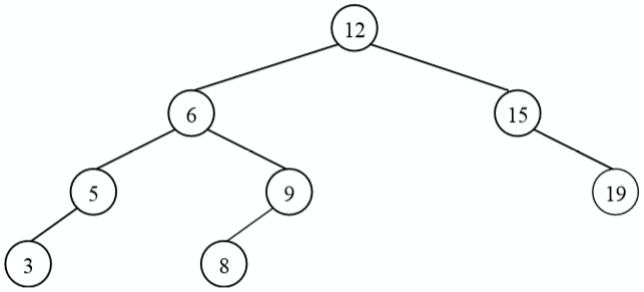


Définition d'un arbre de recherche

Un arbre de recherche est un arbre binaire satisfaisant aux propriétés suivantes :

- L'étiquette appelée clé est de type ordonné
- Pour tout sous-arbre
  - Clés(sous-arbre gauche) < clé(racine)
  - Clés(sous-arbre droite) > clé(racine)

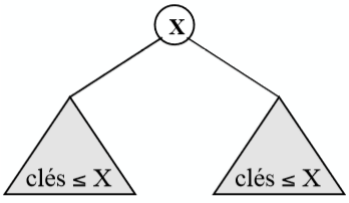




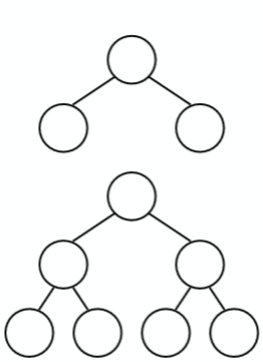
Définition d'un tas

Un tas est un arbre binaire complet + conditions :

- Étiquette de type ordonnée = clé
- Clé(racine) ≥ clé(sous-arbre gauche)
- Clé(racine) ≤ clé(sous-arbre droit)



Hauteur d'un arbre binaire complet



n	h (hauteur)
3	2
7	3

$$n = 2^h - 1$$

$$h = \log_2(n+1)$$

Complexités d'algorithmes pour arbre de recherche

Algorithme	Complexité en moyenne	Complexité dans le pire des cas
Recherche	O(log(n))	O(n)
Insertion	O(log(n))	O(n)
Suppression	O(log(n))	O(n)
Parcours	O(n)	O(n)
Tri par arbre de recherche	O(n · log(n))	O(n · n)

Equilibrage des arbres de recherche

Il y a deux approches :

- Équilibrage de tout l'arbre sur demande
- Rééquilibrage automatique lors de chaque modification

Algorithme :

```
linéariser l'arbre de recherche en tenant comptant les n nœuds
arboriser l'arbre à partir des n nœuds de la liste
```

Linéariser :

```
si l'arbre n'est pas vide alors
    linéariser le sous-arbre droit en utilisant la liste
    insérer la racine de l'arbre en tête de liste
    incrémenter de 1 le nombre d'éléments
    linéariser la sous-arbre gauche en utilisant la liste
fin si
```

Arboriser :

```
si le nombre de nœuds n est > 0 alors
    arboriser un sous-arbre gauche à partir des (n-1)/2 (premiers)
       noeuds de la liste
    extraire le premier nœud restant dans la liste et en faire la
       racine de l'arbre
    arboriser un sous-arbre droit à partir des n/2 nœud restants de
       la liste
    accrocher le sous-arbre gauche à gauche de la racine
    accrocher le sous-arbre droit à droit de la racine
fin si
```

Définition d'un arbre binaire équilibré

Un arbre est équilibré si pour chacun de ses sous-arbre :

|hauteur(sous-arbre gauche) – hauteur(sous-arbre droit)| ≤ 1

Hachage

Motivation

Une information complète se retrouve normalement à l'aide d'une clé qui l'identifie.

Nous voulons un ensemble dynamique d'informations, c-à-d aussi pouvoir ajouter ou supprimer un élément d'information. On en vient naturellement à définir un type abstrait de données, appelée table d'association qui offre les opérations suivantes :

- Trouver l'information associée une clé donnée
- Ajouter une nouvelle association entre une clé et une information
- Retirer une clé de la table (avec les informations associée)

On peut retrouver rapidement une information dans une grande quantité d'informations avec les méthodes suivantes :

- Recherche séquentielle O(n)
- Arbre de recherche équilibré O(log(n))
- Recherche par dichotomie O(log(n))
- Recherche par fonction de hachage O(1)

Algorithme de recherche

Séquentielle

```
pour chaque ligne de la table boucler
   si la clé cherchée = clé de cette ligne alors
      retourner la ligne (elle contient la clé cherchée)
   fin si
fin boucler
signaler qu'aucune ligne ne contient la clé cherchée
```

Dichotomie

```
affecter à la variable premier la valeur d'indice de la première ligne du tableau
affecter à la variable dernier la valeur d'indice de la dernière ligne du tableau
tant que premier <= dernier boucler
        déterminer la clé de la ligne située au milieu de l'intervalle
        si la clé cherchée = cette clé alors retourner la ligne (elle contient la clé cherchée)
        sinon si la clé recherchée < cette clé alors affecter à dernier l'indice (de la ligne du milieu) moins 1
        sinon (la clé recherchée > cette clé affecter à premier l'indice (de la ligne du milieu) plus 1
        fin si
fin boucler
Signaler qu'aucune ligne ne contient la clé recherchée
```

Un arbre de recherche

```
si l'arbre est vide alors
   signaler qu'aucune ligne ne contient la clé cherchée
sinon
   si la clé de la ligne recherchée < la racine alors
      rechercher la ligne dans le sous-arbre gauche
   si la clé de la ligne recherchée > la racine alors
      rechercher la ligne dans le sous-arbre droit
   sinon
      retourner la ligne de la racine (clé racine = clé cherchée)
   fin si
fin si
```

Inconvénients des algorithmes précédents

- La recherche séquentielle est la plus lente
- La dichotomie implique un tri, par exemple, après chaque insertion
- L'arbre de recherche doit être rééquilibré si nécessaire

Table de hachage, fonction de hachage

Les méthodes de hachage tentent de trouver une information en calculant sa position à partir de sa clé. Les informations sont stockées dans un tableau linéaire avec des indices 0, 1, 2, …, m-1 qu'on appelle la table de hachage.

La fonction de hachage est une application h : k -> {0, 1, 2, .., m-1} qui fait correspondre à chaque clé k, un indice h(k), appelé l'adresse de hachage.

Donc pour savoir où stocker un élément avec une clé k dans la table de hachage, il suffit de calculer h(k) qui nous donnera l'indice du tableau où insérer l'élément.

Pour rechercher un élément, on calcule son adresse de hachage h(k) puis on regarde la position correspondante dans la table de hachage. Cependant, m est en général beaucoup plus petit que | K |. La fonction de hachage n'est donc en général pas injective, c-à-d que plusieurs clés peuvent avoir la même adresse de hachage.

Collision

Soit h une fonction de hachage et soient k, k' ∈ K. Si k ≠k' et h(k) = h(k'), alors on dit qu'on a une collision d'adresse, et k et k' sont appelées des synonymes.

Une bonne méthode de hachage doit avoir les propriétés suivantes :

- Elle doit donner aussi peu de collision d'adresse que possible
- Les collisions doivent être résolues de manière efficace
- Il doit être possible de calculer l'indice de hachage h(k) d'une clé de manière efficace

Caractéristique d'une bonne fonction de hachage

Elle doit :

- Minimiser les collisions
- Etre facile à calculer
- Distribuer les valeurs associées aux clés uniformément dans la table de hachage
- Utiliser toute l'information fournie par la clé
- Avoir un taux élevé du taux d'occupation pour un ensemble de clés donné

Le taux d'occupation d'une table de hachage non vide représente le rapport entre le nombre d'éléments stockés dans la table et la taille de la table. C'est le paramètre qui nous permet de décider à re-hacher ou à redimensionner la table. Il nous informe si la distribution des clés est uniforme ou pas :

$$\text{Taux d'occupation} = \frac{\text{Nombre d'éléments dans la table de hachage}}{\text{Taille de la table de hachage}}$$

Lorsque le facteur de compression de la table augmente au-delà de 50%, le nombre de collisions augmente sensiblement. Une solution est d'augmenter la taille de la table sitôt atteint ce taux, tout en maintenant cette taille à un nombre premier. Rehachage est une fonction qui, en général double l'espace mémoire alloué pour la table, et recopie ses valeurs. Cette fonction cherche le plus petit nombre premier supérieur à 2 fois sa taille.

Adressage

Extraction

Dans une fonction de hachage avec adressage par extraction, on extrait des bits de la clé pour obtenir la valeur de hachage.

Ex : avec les bits 3, 10, 18 et 23 et un codage des bits par entier *h*(hello)= 110110 110000 110010 110010 110101= 1011 =11

Inconvénient, la valeur de hachage ne dépend pas de l'intégralité de la clé.

h(hello) = *h*(Say hello)= 11

Une bonne fonction de hachage doit faire intervenir tous les bits de la clé.

Division

Hachage avec adressage par division :

- *h: K → {0,...,m-1}, h(k) = k modulo m*
- Ex : avec *m*=5, *h*(13)=3

Cette technique a une bonne répartition mais multiplie les collisions. On la combinera avec d'autre méthode.

Pour minimiser les collisions on doit choisir : m premier et éloigné des puissances de 2.

Multiplication

Hachage avec adressage par multiplication :

$h: K \rightarrow \{0,...,m-1\}, h(k) = \lfloor m \cdot (k \cdot A \text{ modulo } 1) \rfloor$ avec $A=cst$

Le choix de la constante de A dépend du type de la clé.

Multiplication, addition et division

Hachage avec adressage par Multiplication, addition et division (MAD).

$h: K \rightarrow \{0,...,m-1\}$
$h(k) = (ak + b) \text{ modulo } m$, avec $a > 0$ et $b > 0$

Compression

Hachage avec adressage par compression :

- On découpe en morceaux de n bits le code de la clé
- On compresse les morceaux avec une opération : addition, et, ou, ou exclusif et polynomiale

Compression polynomiale

Dans la compression polynomiale :

- Les sous chaînes ont des longueurs de 8, 16 ou 32 bits
- Chaque sous chaîne représente un coefficient
- On calcule le polynôme
  - $P(z)=a_0 + a_1 z + a_2 z^2 + a_3 z^3 + ... + a_{n-1} z^{n-1}$

Très bonne méthode pour le type string.

Résolution des collisions

On ne peut pas éviter les collisions. Pour résoudre les collisions il y a deux méthodes :

Le chaînage

Les données correspondant à une même valeur de hachage sont stockées dans une liste. La fonction de hachage ne retourne qu'une valeur par clé et on range la clé à parti de cette valeur.

L'adressage ouvert

On place les données dans la première place libre dans le tableau. La fonction de hachage retourne un ensemble de valeurs. Cet ensemble représente les rangements possibles pour cette clé. Cet ensemble sera parcouru pour l'insertion comme pour la recherche.

Résolution par chaînage

Les éléments en collisions sont chaînés entre eux à l'extérieur du tableau de hachage. La table de hachage contient le début de ces listes chainées.

```
calcul de la valeur de hachage : h(k) →@
si tab[@] contient la clé k alors
      succès
sinon
      explore la liste jusqu'à trouver la clé k
fin si
```

Les avantages sont :

- Recherche facile
  - Chaque case de la table est en fait une liste chaînée des clés qui ont le même hachage. Une fois la case trouvée, la recherche est alors linéaire en la taille de la liste chaînée. Dans le pire des cas où la fonction de hachage renvoie toujours la même valeur de hachage quelle que soit la clé, la table de hachage devient alors une liste chaînée, et le temps de recherche est en O(n).
- Suppression facile
- Seul l'ordre des clés en collisions compte

Les inconvénients sont :

- Une allocation mémoire à chaque enregistrement
- Taille : un pointeur supplémentaire pour chaque clé

Résolution par coalescence

Principe de la résolution par coalescence :

Lorsqu'il y a collision avec une autre clé on cherche une autre place disponible. On note cette place comme le NEXT de la première clé. Au fil des collisions, ces liens explicites forment une liste chaînée dans la table.

```
calcul de la valeur de hachage : h(k) → @
tant que tab[@] ne contient pas la clé k boucler
   si next[@] = -1 alors
      echec
   sinon
      @ ←next[@]
   fin si
 fin boucler
 succès
```

Les avantages :

- Pas d'allocation mémoire pour un enregistrement

Les inconvénients :

- Le nombre de clé est limité à la taille de la table
- Taille : un lien supplémentaire pour chaque clé
- La table dépend de l'ordre de toutes les clés
- Suppression très compliquée

Résolution par sondage linéaire

Principe par sondage linéaire :

Lorsqu'il y a collision avec une autre clé on parcourt circulairement la table pour chercher une place disponible

```
calcul de la valeur de hachage : h(k) → @ → @initiale
tanque tab[@] ne contient pas la clé k faire
   @←(@ + 1) mod m
   si @ = @initiale alors
      echec
fin tanque
succès
```

Les avantages :

- Pas d'allocation mémoire pour un enregistrement
- Recherche fructueuse facile
- Suppression facile
- Taille : réduite au minimum

Les inconvénients :

- Recherche fructueuse en O(n)
- Le nombre de clé est limité à la taille de la table
- La table dépend de l'ordre de toutes les clés
- Formation d'amas dans la table

Résolution par double-hachage

Principe par double-hachage :

Lorsqu'il y a collision avec une autre clé on parcourt circulairement la table, avec un pas donné par une autre fonction de hachage, pour chercher une place disponible.

```
calcul de la valeur de hachage : h1(k) → @
tanque tab[@] ne contient pas la clé k faire
   @←(@ + h2(k)) mod m
   si @ déjà parcouru alors
      echec
fin tanque
succès
```

Pour s'assurer de bien parcourir toute la table :

$h_2(k)$ doit être premier avec m (le nombre de case)

Les avantages :

- Pas d'allocation mémoire pour un enregistrement
- Recherche fructueuse facile
- Suppression facile
- Taille : réduite au minimum
- Evite la formation d'amas dans la table

Les inconvénients :

- Recherche infructueuse en O(n)
- Le nombre de clé est limité à la taille de la table
- La table dépend de l'ordre de toutes les clés

Conclusion

Les tables de hachage se résument à un compromis entre espace mémoire et collisions :

- Avec un mémoire infinie : on évite toutes les collisions
- Avec un mémoire minimum : toutes les clés sont en collision. Elles sont rangées dans une liste