Introduction

Langage de description algorithmique

Complexité

Structures de données

Types abstraits de données

Séquences

Listes

Listes itératives

Listes récursives

Piles

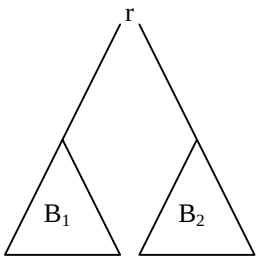
Files

Les structures arborescentes

Un *arbre* est un ensemble d'éléments appelés *nœuds* ou *sommets* organisés de manière hiérarchique à partir d'un nœud distingué appelé *racine*. On repère les nœuds de l'arbre en leur donnant des noms différents.

Arbres binaires

Un *arbre binaire* est soit *vide*, noté Ø, soit de la forme < r, B_1, B_2 > où r est la racine et où B_1 et B_2 sont des arbres binaires.



On définit intuitivement les notions de *sous-arbre*, *de sous-arbre gauche*, *de fils gauche*, *de lien gauche* et réciproquement à droite. On définit encore les notions de *père*, de *frère*, *d'ascendant* et de *descendant*. Les nœuds d'un arbre binaire ont au plus deux fils. Un *nœud interne ou double* a 2 fils. Un *point simple à gauche* a seulement un fils à gauche, et réciproquement à droite. Un *nœud externe ou feuille* n'a pas de fils. De façon généralisé, on qualifie de *nœud interne*, tout nœud qui n'est pas externe. On appelle les *branches* de l'arbre tout *chemin* (suite consécutive de nœuds) allant de la racine à une feuille. Il y a autant de branches que de feuilles. Le *bord gauche* est le chemin issu de la racine en suivant les liens gauches, et réciproquement à droite.

Les caractéristiques d'un arbre sont :

- *la taille* de l'arbre est le nombre total de nœuds.
- *la hauteur ou niveau d'un nœud x* est le nombre de lien sur l'unique chemin allant de la racine à *x*, notée *h(x).*
- *la hauteur ou profondeur de l'arbre A,* $h(A) = \max_x \{h(x)\}$.

- *la longueur de cheminement de l'arbre A,* $LC(A)=\sum_{x} h(x)=LCE(A)+LCI(A)$ avec *LCE* *la longueur de cheminement extérieure* $\sum_{x \text{ feuille}} h(x)$ et *LCI la longueur de cheminement intérieure* $\sum_{x \text{ noeud interne}} h(x)$.
- *la profondeur moyenne externe* de *A* est $PE(A)=\frac{LCE(A)}{\text{nombre de feuilles}}$.

On donne la signature du type abstrait arbre binaire :

sorte arbre
utilise nœud
opérations
Ø : → arbre
< – , – , – > : nœud x arbre x arbre → arbre
racine : arbre → nœud
gauche : arbre → arbre
droit : arbre → arbre.

Proposition : Soit un arbre binaire de taille *n*, de profondeur *p* et possédant f feuilles. Si on examine les cas extrêmes, on obtient que $\lfloor \log_2(n) \rfloor \leq p \leq n-1$; par ailleurs $f \leq 2^p$ d'où $p \geq \lceil \log_2(f) \rceil$.

Représentation des arbres en machines : On utilise la structure récursive des arbres.
1) Dans cette première représentation, le chaînage est symbolisé par des pointeurs.

```
type arbre = adresse de nœud ;
type nœud = structure
            val : élément ;
            g, d : arbre ;
            fin ;
```

L'arbre est déterminé par l'adresse de la racine. On gère la mémoire dynamiquement pour les opérations de suppression / insertion.

2) Dans *la représentation contiguë,* on simule les chaînages par des indices dans un tableau.

```
type arbre = tableau de 1 à N de
                structure
                val : élément ;
                g, d : entier ;
                fin ;
```

On utilise l'indice 0 pour indiquer qu'il n'y a pas de fils. L'arbre est déterminé par une variable entière contenant l'indice de la racine. On va gérer les cases libres pour des insertions / suppressions. On chaîne les cases libres comme une pile, en utilisant le chaînage *g* et on utilise une variable entière *libre* pour indiquer l'indice du « sommet de pile ».

Arbres binaires complets

Un arbre binaire est *complet* si tous les nœuds qui ne sont pas des feuilles ont 2 fils.

Propositions : Un arbre binaire complet ayant *n* nœuds internes possède en tout *n+1* feuilles. La démonstration se fait simplement par récurrence. Par ailleurs, on montre qu'il existe une bijection entre l'ensemble des arbres binaires de tailles n, B_n, et l'ensemble des arbres binaires complets de taille *2n+1*, BC_{2n+1}. Principe : on ajoute des feuilles de sorte que tout nœud de l'arbre ait deux fils. De plus on établit que $|B_n| = |BC_{2n+1}| = \frac{1}{n+1} C_{2n}^{n}$.

Arbres binaires parfaits, ordre hiérarchique

On dit qu'un arbre binaire est *parfait* si toutes ses feuilles sont situées sur les deux derniers niveau, l'avant dernier étant complet, et les feuilles du dernier sont le plus à gauche possible. Attention ! un arbre binaire parfait n'est pas forcément complet, mais il a toujours au plus un nœud simple (le père de la feuille la plus à droite).

On peut représenter un arbre binaire parfait de taille n de manière compacte par un tableau à n cases. Ceci se fait en numérotant les nœuds de 1 à n en partant de la racine, niveau par niveau de gauche à droite (ordre hièrarchique). On a les relation générales suivantes :

- le père du nœud d'indice i est à l'indice i / 2 (division entière) ;
- le fils gauche d'un nœud i est, s'il existe, à l'indice 2i ;
- le fils droit d'un nœud i est, s'il existe, à l'indice 2i+1 ;
- les feuilles sont aux indices > n / 2.

Parcours en profondeur d'un arbre binaire

On considère l'opération de *parcours* qui consiste à examiner systématiquement dans un certain ordre tous les nœuds de l'arbres pour effectuer un traitement de données. *Le parcours en profondeur à gauche* consiste à partir de la racine et à tourner autour de l'arbre en allant toujours le plus à gauche possible.

```
Procédure Parcours(A : arbre) ;
début
Si A = Ø
        Alors T0
        Sinon
                début
                T1 ;
                Parcours(g(A)) ;
                T2 ;
                Parcours(d(A)) ;
                T3 ;
                fin ;
fin ;
```

Chaque nœud est visité trois fois. A chaque visite correspond un traitement T_i et un ordre de parcours. T_1 s'effectue avant la descente gauche et décrit *l'ordre préfixe* ou *pré-ordre*. T_2 s'effectue après la remontée gauche et avant la remontée droite, l'ordre associé est *l'ordre infixe* ou *symétrique.* Le traitement T_3 est réalisé après la remontée droite ; les nœuds sont parcourus dans *l'ordre suffixe* ou *post-fixe*. On ajoute un traitement particulier T_0 pour les nœuds vides.

Arbres généraux

Un *arbre général*, ou simplement *arbre*, est une structure arborescente où le nombre de fils n'est plus limité à 2. Un arbre $A = < r, A_1, A_2, \ldots, A_n >$ est la donnée d'une racine r et d'une suite éventuellement vide de *sous-arbres disjoints*. Cette suite est une *forêt*. Un arbre est obtenu en rajoutant un nœud racine à la forêt.

On donne la signature des arbres généraux :

```
sorte arbre, forêt
utilise nœud
opérations
        cons : nœud x forêt → arbre
        racine : arbre → nœud
        sous-arbre : arbre → forêt
        Ø : → forêt
        ième : forêt x entier → arbre
        longueur : forêt → entier
        insérer : forêt x entier x arbre → forêt
```

Il n'y a plus de notion de fils gauche ou de fils droit. On parle du *ième fils* d'un nœud.

Dans un parcours à gauche, chaque nœud est rencontré une fois de plus que son nombre de fils.

```
Procédure Parcours(A : arbre) ;
début
Si sous-arbre(A) = Ø
        Alors T0
        Sinon
                début
                i ← 1 ;
                Répéter
                        Ti ;
                        Parcours(ième(sous-arbre(A), i)) ;
                        i++ ;
                Jusqu'à ce que i > longueur(sous-arbre(A)) ;
                Ti ;
                fin ;
fin ;
```

L'ordre *préfixe* sur les nœuds est obtenu en ne faisant qu'intervenir que T_1. L'ordre *suffixe* est obtenu en ne faisant intervenir que T_i à l'extérieur de la boucle. Il n'y a pas d'ordre *infixe*.

Représentation des arbres : On représente un arbre général par des listes chaînées dynamiques.

```
type arbre = adresse de nœud ;
type nœud = structure
        val : élément ;
        premier_fils, frère : arbre ;
        fin ;
```

Propositions : Le nombre d'arbres généraux de taille n+1 est $\frac{1}{n+1}C_{2n}^{n}$. Par ailleurs, il existe des bijections entre les forêts de taille n, les arbres généraux de taille n+1, et les arbres binaires de taille n, avec des propriétés intéressantes sur les parcours.

Graphes

Définition

Un *graphe* est un ensemble d'objets modélisés par des *sommets*, et mis en relation (binaire). Ces relations sont modélisés par des *arcs* ou des *arêtes*. Un graphe *orienté* [*non orienté*] est un couple $G = \langle S, A \rangle$ où S est un ensemble fini de sommets, et A un ensemble de paire ordonnées [couple] de sommets appelés arcs [arêtes].

Terminologie

Un *graphe simple* est sans *boucle*[1], et sans *liens multiples*. Dans un graphe orienté, on dit que *y* est le *successeur* de *x* s'il existe un arc qui mène de *x* vers *y* ; on dit en outre que *y* est *adjacent* à *x*. Pour un graphe orienté, on dit simplement que *x* et *y* sont *adjacents*. Un graphe est dit *complet* si pour tout couple de sommet il existe un arc (ou une arête) les joignant. Dans un graphe orienté, le *demi-degré extérieur* [*intérieur*] d'un sommet *x*, que l'on note $d^+(x)$ [$d^-(x)$], est le nombre d'arcs ayant *x* comme extrémité initiale (finale). Le *degré* de *x* est $d(x) = d^+(x) + d^-(x)$. Pour un graphe non orienté, on définit uniquement le *degré* d'un sommet *x* $d(x)$. Dans un graphe orienté, on appelle *chemin* de longueur *L* une suite de *L+1* sommets $(s_0, s_1 \cdots s_L)$ telles que (s_i, s_{i+1}) forme un arc. Pour un graphe non orienté, on parle de *chaîne*. Dans un graphe orienté [non orienté], un chemin [une chaîne] dont tous les arcs [arêtes] sont distincts et tels que les sommets aux extrémités coïncident est un *circuit* [un *cycle*]. Un graphe orienté est *fortement connexe* si pour toute paire de sommets distincts *s* et *s'*, il existe un chemin de *s* vers *s'* et un chemin de *s'* vers *s*. Un graphe non orienté est *connexe*, si pour toute paire

[1] lien d'un sommet sur lui-même

de sommets distincts, il existe une chaîne les joignant. Une *composante fortement connexe* [*connexe*] est un sous-graphe fortement connexe [connexe] maximal.

Graphe et Arbre

En théorie des graphes, un arbre est un graphe non orienté, connexe et sans cycle. Soit G un graphe orienté, on appelle *racine* de G un sommet *r* tel que, pour tous sommets *x* distincts de *r*, il existe un chemin de *r* vers *x*. Une *arborescence* est un graphe orienté G admettant une racine et tel que le graphe obtenu à partir de G en enlevant l'orientation soit un arbre.

Signature graphe orienté

sorte sommet
utilise booléen, entier
opérations
s : entier → sommet
n° : sommet → entier
– arc – : sommet x sommet → booléen
d^+ : sommet → entier
ième_succ : sommet x entier → sommet
prem_succ : sommet → sommet
succ_suivant : sommet x sommet → sommet

Pour les graphes non orientés, on dispose de la même signature en remplaçant – *arc* – par – *arête* – et d^+ par *d*.

Représentation des graphes

On aura deux possibilités classiques de gestion de la mémoire : *contiguë* et *chaînée*.

Représentation contiguë : Soit n le nombre de sommet d'un graphe ; on définit *la matrice d'incidence* de dimension *n x n* noté *G* et tel que $G[i, j] = vrai$ ssi il existe un arc de *i* vers *j*. Si le graphe est non orienté la matrice d'incidence est symétrique.

type graphe = tableau[1 à n, 1 à n] de booléen.

La complexité en espace est en $O(n^2)$, parcourir les successeurs d'un sommet se fait en $O(n)$, savoir si *y* est le successeur de *x* se fait en $O(1)$.

Représentation chaînée : Pour chaque sommet s_i, on forme *la liste d'adjacence*, qui est la liste chaînée de tous le successeur de s_i.

type graphe = tableau[1 à n] d'adresse de cellule;
cellule = structure
numéro : entier de 1 à n;
suivant : adresse de cellule;
fin;

Soit $|A| = \sum_x d^+(x)$. La complexité en espace est en $n + 2p$. Le parcours des successeurs d'un sommet s'effectue en $O(d^+(x))$. La consultation complète est en $O(|A|)$. Savoir si *y* est le successeur de *x* se fait en $O(d^+(x))$, dans le pire des cas.

Remarque. Le chaînage peut être simulé dans un tableau. Pour un graphe non orienté, il y a redondance[2] d'information.

[2] y est le successeur de x et réciproquement

Parcours en profondeur d'un graphe orienté

Le *parcours en profondeur* un parcours récursif, simulable par une *pile*.
Principe : On utilise une marque (vecteur de n booléens) pour marquer les sommets au fur et à mesure qu'on les rencontre. A partir d'un *x* de départ, non marqué, on avance dans le graphe en allant toujours vers un nouveau successeur non marqué ; les sommets retenus à chaque fois sont marqués. Quand on ne peut plus avancer, on revient au choix précédent, et on itère la méthode.

```
Procédure Profondeur(x : sommet) ;
début
i ← n°(x) ;
marque[i] ← vrai ;
pour j de 1 à d+(x) faire
        début
        y ← ième_succ(x, j) ;
        k ← n°(y) ;
        si non marque[k] alors Profondeur(y) ;
        fin ;
fin_Profondeur ;

Programme principal
début
pour i de 1 à n faire marque[i] ← faux ;
pour i de 1 à n faire
        si non marque[i] alors profondeur(s(i)) ;
fin.
```

Pour les graphes non orientés, on dispose du même algorithme en remplaçant $d^+(x)$ par d.

Les sommets ne sont marqués qu'une seule fois et l'algorithme parcourt un fois et une seule les listes d'adjacence : complexité en $\sum_x d^+(x)$, ce qui donne $O(|A|)$ pour les listes d'adjacence et $O(n^2)$ pour les matrices d'adjacence.

On considère les arcs $x \rightarrow y$ tels que *Profondeur(x)* appelle *Profondeur(y)*. Ces *arcs couvrants* constituent *une forêt couvrante* constituée *d'arborescences* disjointes et dont les racines sont les sommets de départ. Les graphes obtenu sans orientation sont des arbres (graphe non orienté, connexe et sans cycle). La *forêt couvrante* a autant *d'arbres recouvrants* qu'il y a de composantes connexes dans le graphe. Ainsi le parcours en profondeur résout le test de connexité en temps linéaire.

Parcours en largeur

Le *parcours par largeur ou par niveau* est un parcours itératif qui fonctionne avec une *file*.
Principe : On part d'un sommet *x* et on visite tous les successeurs *y* de *x* ; on réitère l'algorithme sur les sommets y dans l'ordre où on les a rencontrés à partir de *x*. On utilise toujours une marque de sommets. On utilise une file pour gérer ce parcours par niveaux.

```
Procédure Largeur(x :sommet)
début
file_vide(file) ;
i ← n°(x) ;
marque[i] ← vraie ;
ajouter(file, x) ;
tant que non est_vide(file) faire
        début
        y ← premier(file) ;
        retirer(file) ;
        pour i de 1 à d+(y) faire
                début
                z ← ième_succ(y,i) ;
                j ← n°(z);
```

```
                si non marque[j] alors
                        début
                        marque[j] ← vraie;
                        ajouter(file,z);
                        fin;

                fin ;
        fin ;
fin ;

Programme principal
début
pour i de 1 à n faire marque[i] ← faux ;
pour i de 1 à n faire
        si non marque[i] alors Largeur(s(i)) ;
fin.
```

On a la même complexité que pour le parcours en profondeur. L'algorithme pour un graphe non orienté s'obtient simplement en remplaçant d^+ par d. On a la même propriété sur la forêt couvrante et les composantes connexes que pour le parcours en profondeur.

Problème de la recherche

Introduction

Considérons un ensemble de grande taille ayant des caractéristiques communes. On veut faire de manière optimisée des opérations de *recherche*, d'*adjonction* et de *suppression*. A chaque élément, on associe une *clé simple* (critère unique) : recherche associative. Les bases de données traitent des critères plus généraux et des clés multiples.

Signature

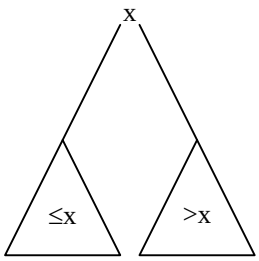
sorte ensemble
utilise élément, clef
opérations
- *clé : élément → clef*
- *vide : → ensemble*
- *ajouter : élément x ensemble → ensemble*
- *supprimer : clef x ensemble → ensemble*
- *_ ∈ _ : clef x ensemble → booléen*

Remarques :
- Si plusieurs éléments ont la même clé, la recherche fournit une solution quelconque parmi les possibilités ; s'il n'y a pas de solutions (échec), on fournit une valeur spéciale.
- La suppression commence par une recherche, en cas d'échec de la recherche, la suppression laisse l'ensemble inchangé.
- En général, et s'il n'y a pas ambiguïté, on confond l'élément et sa *clé*.

La complexité fondamentale est celle de la comparaison entre deux clés. Si l'ensemble des clés est muni d'une *relation d'ordre*, on peut les *trier* avec des algorithmes efficaces. On distingue *les méthodes de recherche séquentielle, les méthodes de recherche dichotomique, de hachage, et les méthodes arborescentes.*

Arbres binaires de recherche

On représente un ensemble ordonné à *n* éléments par un arbre binaire à n nœuds (les nœuds sont les éléments), et c'est la comparaison avec la valeur d'un nœud qui va orienter la suite de la recherche.
Un *arbre binaire de recherche* est un arbre binaire tel que pour tout nœud x , les nœuds de son sous arbre-gauche s'ils en existent ont des valeurs inférieures ou égales à celle de x, et les nœuds de son sous arbre-droit des valeurs strictement supérieures ;



ce que l'on traduit par $g(A) \leq racine(A) < d(A)$.

On obtient toujours l'ensemble ordonné par un parcours récursif gauche symétrique. Il n'y a pas unicité de la représentation.

Recherche d'un élément

rechercher : valeur × arbre → booléen

On compare l'élément à la valeur de la racine :

- égalité → succès

  $x = r \Rightarrow rechercher\ (x\ ,\ <\ r\ ,\ g\ \ ,\ d\ >\ ) = vraie$

- si le sous-arbre sélectionné est vide, l'élément est absent → échec

  $rechercher\ (\ x\ ,\ \varnothing\ ) = faux$

- si la valeur est plus petite, on recommence récurcivement dans le sous-arbre gauche ; et réciproquement si la valeur est plus grande dans le sous-arbre droit

  $x < r \Rightarrow rechercher\ (x\ ,\ <\ r\ ,\ g\ \ ,\ d\ >\ ) = rechercher\ (x\ ,\ \ g\ \ )$
  $x > r \Rightarrow rechercher\ (x\ ,\ <\ r\ ,\ g\ \ ,\ d\ >\ ) = rechercher\ (x\ ,\ \ d\ \ )$

Complexité : La complexité au pire est en *O (hauteur de l'arbre).*

Adjonction d'un élément aux feuilles

ajout_feuille : arbre × valeur → arbre

L'adjonction aux feuilles d'un élément se réalise en deux étapes :
- étape de recherche pour savoir où insérer le nouvel élément ;
- adjonction elle-même.

On compare la valeur de l'élément à ajouter à celle de la racine pour déterminer si on l'ajoute sur le sous-arbre gauche ou droit.

$x \leq r \Rightarrow ajout\_feuille\ (\ <\ r\ ,\ g\ ,\ d\ >\ ,\ x\ ) = <\ r\ ,\ ajout\_feuille\ (\ g\ ,\ x\ )\ ,\ d\ >$
$x > r \Rightarrow ajout\_feuille\ (\ <\ r\ ,\ g\ ,\ d\ >\ ,\ x\ ) = <\ r\ ,\ g\ ,\ ajout\_feuille\ (\ d\ ,\ x\ )\ >$

Le dernier appel récursif se fait sur un arbre vide ; on crée un nouveau nœud à cette place pour le nouvel élément qui devient donc une feuille.

$ajout\_feuille\ (\ \varnothing, x\ ) = <\ x\ ,\ \varnothing,\ \varnothing\ >$

On peut construire un arbre binaire de recherche par adjonctions successives aux feuilles.

On donne l'algorithme d'adjonction aux feuilles (récursif) en LDA :

```
Fonction adjonction_feuille (A : arbre , e : entier ) : arbre
début
si est_vide(A)
        alors retourner < e , ∅, ∅ >
sinon si ( e ≤ racine(A) )
        retourner < racine(A) , ajout_feuille( g(A) , e ) , d(A) >
sinon
        retourner < racine(A) ,g(A) ,  ajout_feuille( d(A) , e ) >
fin
```

On donne également une version itérative de l'algorithme (→ voire td…).

La *complexité* d'un adjonction est *O (hauteur de l'arbre).*

Adjonction d'un élément à la racine

Soit A un arbre binaire de recherche, on veut ajouter *x* à la racine.

On procède en deux étapes :

- on coupe A en deux arbres binaires de recherche G et D contenant respectivement tous les éléments $\leq x$ et tous ceux $> x$.
- construire l'arbre $< x , G , D >$

→ voire cours…

Suppression d'un élément

supprimer : arbre × valeur → arbre

- recherche de l'élément à supprimer
- suppression qui dépend de la place de l'élément, selon que le nœud est sans fils, avec un seul fils, ou avec deux fils. La suppression d'un nœud sans fils est immédiate. Pour la suppression un nœud avec un seul fils, on remplace ce nœud par son fils. Pour un nœud, avec deux fils, on dispose de deux solutions : soit on remplace le nœud à supprimer par le plus grand élément de son sous-arbre gauche, soit on le remplace par le plus petit élément de son sous-arbre droit.

On donne l'algorithme de suppression en LDA :

```
Fonction suppression ( A : arbre , e : entier ) : arbre
début
% recherche de l'élément à supprimer %
si est_vide(A)
        alors retourner erreur
si ( e < racine(A) )
        alors retourner < racine(A), suppression( g(A) , e ) , d(A) )
sinon si ( e > racine(A) )
        retourner < racine(A), g(A) , suppression( d(A) , e ) )
% suppression %
sinon
        si est_feuille(A)
                alors retourner ∅
        sinon si est_vide(g(A))
                retourner d(A)
        sinon si est_vide(d(A))
                retourner g(A)
        sinon
                %  on ajoute l'élément le plus à droite du sous-arbre gauche %
                retourner < max_noeud(g(A)) , retire_max(g(A)), d(A) >
```

```
fin

Fonction max_noeud ( A : arbre ) : entier
% retourne le plus grand élément de l'arbre A, le plus à droite %
début
si est_vide(d(A))
        alors retourner racine(A)
sinon
        retourner max(d(A))
fin

Fonction retire_max ( A : arbre ) : arbre
% retourne l'arbre privé de son plus grand élément %
début
si est_vide(d(A))
        alors retourner g(A)
sinon
        retourner < racine(A) , g(A) , retire_ max(d(A)) >
fin
```

La complexité est *O (hauteur de l'arbre)*.

Conclusion, Tri par arbre binaire de recherche

Toutes les opérations ont une complexité dépendant de la hauteur de l'arbre binaire de recherche. Elle varie entre *O (log n)* pour des arbres équilibrés et *O (n)* pour des arbres dégénérés.

Par conséquent, *le tri par arbre binaire de recherche*, obtenu par un parcours symétrique de l'arbre, a une complexité en comparaison dans le pire des cas variant entre *O (n log n)* et *O (n²).*

Problème du tri

Introduction

Le problème du tri est quasiment le plus important en informatique.

Spécification du tri : La donnée est une liste à n éléments ; à chaque élément est associée une clé dont la valeur appartient à un ensemble totalement ordonné ; le résultat est une liste dont les éléments sont une permutation de la liste d'origine, et telle que les valeurs des clés soient croissantes quand on parcourt la liste séquentiellement.

Un tri est *stable*, s'il conserve l'ordre de départ des éléments dont les clés sont égales.

En fonction de la capacité mémoire, on distingue *le tri interne* (tout en mémoire centrale) et *le tri externe* (mémoire centrale + disque). Pour le tri interne, on a des algorithmes qui travaillent *sur place*, c'est-à-dire sur la liste de départ et des algorithmes qui manipulent physiquement une copie. On a des algorithmes différents et plus compliqués quand ils se font sur place.

On compte le nombre de variables auxiliaires pour évaluer *la complexité en mémoire.* Le tri interne, sur place, avec un nombre constant de variables auxiliaires possède une bonne complexité en espace. On compte le nombre de comparaisons entre clés et de transferts d'éléments pour évaluer *la complexité en temps*.

On distingue *les méthodes simples* et les *méthodes plus complexes.*

Tri à bulle

…

Tri par insertion

…

Tri par arbre binaire de recherche

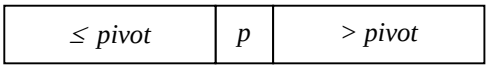
C'est une méthode plus complexe, qui consiste à créer l'arbre binaire de recherche, puis à faire un parcours symétrique, pour obtenir la liste trieé.

→ voire partie sur les arbres binaires de recherche…

Quicksort

Principe

On choisit une des clés de la liste (par exemple, celle du premier élément), et on l'utilise comme *pivot.* On construit *sur place* une sous-liste dont les clés sont inférieures ou égales au pivot et une sous-liste dont les clés sont strictement supérieurs au pivot.



Le pivot *p* a alors sa place définitive. Et on recommence récursivement sur chacune des deux sous-listes. A la fin, la liste est triée par ordre croissant. Remarquons que le choix du pivot est délicat ; de lui dépend l'équilibrage des deux sous listes.

On suppose donnée une procédure

Placer (e/s t : tableau de 1 à n entiers , e/ i : entier , e/ j : entier , /s k : entier)

qui effectue le traitement de *t* entre les indices *i* et *j* en fonction du pivot *t[i]*, et qui rend comme résultat l'indice k où le pivot a été placé et le tableau *t* réagencé.

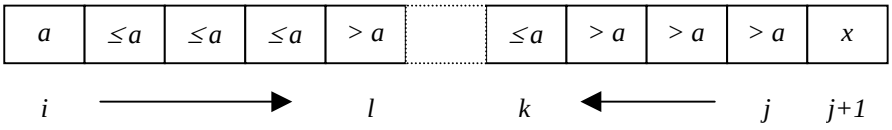
La procédure générique du *quicksort* s'écrit :

```
Procédure Quicksort (e/s t : tableau de 1 à n entiers , e/ i : entier , e/ j : entier)
utilise localement k : entier
début
si i < j
        alors début
        Placer (t , i , j , k)
        Quicksort (t , i , k - 1)
        Quicksort (t , k + 1 , j)
        fin

fin
```

Le tri de la liste complète est obtenu par *Quicksort (t , 1 , n).*

La procédure Placer : La partition et le placement du pivot ne nécessite qu'un parcours.



On utilise deux compteurs *l* et *k* qui partent des extrémités du sous-tableau, et qui vont l'un vers l'autre :

- *l* part de *i+1* et avance tant que l'on rencontre un élément ≤ à *a*.
- *k* part de *j* et recule tant que l'on rencontre un élément > à *a*.

On échange les deux éléments et on recommence la progression jusqu'à ce que les deux compteurs se croisent : la place définitive du pivot est en *k*, et on y place le pivot *a* par échange avec un élément ≤ à *a*.

Si on utilise la procédure *Placer* sur un sous-tableau qui n'est pas à la fin de (*x = t[j+1]* existe), alors l'élément *x* est un pivot placé antérieurement. Donc, on a $\forall s \in [i, j], x \geq t[s]$. Par conséquent, cet élément *x* va arrêter la progression du compteur *l.* Pour utiliser cette propriété (effet de bord) lors de tous les appels, *on rajoute un terme en t[n+1]* qui contiendra un élément plus grand que tous les autres.

```
Procédure Placer (e/s  t : tableau de 1 à n entiers , e/  i : entier , e/  j : entier ,  /s k : entier)
utilise localement l : entier
début
l ← i +1
k ← j
% boucle : tant que les compteurs ne se croisent pas %
tant que l ≤ k faire
        début
        tant que t[k] > t[i]  faire k--
        tant que t[l] ≤ t[i]  faire l++
        % on a t[k] ≤ t[i] et t[l] > t[i] %
        si l < k alors début
                échanger t[l] et t[k]
                l++
                k--
                fin

        fin
% on met le pivot à sa place définitive %
échanger t[i] et t[k]
fin
```

Complexité

Complexité de Placer : Considérons un sous-vecteur à *p* éléments, on la pivot aux *p - 1* autres éléments. On effectue en tout *p + 1* comparaisons.

Complexité du Quicksort, au pire: Le graphe des appels du Quicksort est un arbre binaire. La complexité au pire en nombre de comparaisons est obtenu pour *t* déjà trié est en prenant à chaque fois le 1er élément comme pivot. Le graphe des appels sera dégénéré (peigne) et va induire une complexité au pire en $O(n^2)$.

Complexité du Quicksort, en moyenne : On suppose que les *n* nombres sont distincts, et que le pivot va occuper la p^{ième} place de la liste à trier. On suppose que toutes les places sont équiprobables ; on a la probabilité *1/n* d'avoir le pivot à la place *p* et donc d'avoir deux sous-listes de taille *p - 1* et *n - p*. On démontre (→ voire cours + td) que la complexité en moyenne est en $O(2n \log n)$.

Taille de la pile de récursivité

Dans la procédure Quicksort, le 2ème appel est terminal, ce qui veut dire qu'on peut le supprimer, et donc éviter des empilements. Comme un seul appel va être conservé, on l'effectuera systématiquement sur la plus petite des deux sous-listes. La taille de récursion sera en $O(\log_2 n)$, car on divisera par 2 la taille de la liste d'appel.

```
Procédure Quicksort (e/s  t : tableau de 1 à n+1 entiers , e/  i : entier , e/  j : entier)
utilise localement k : entier
début
tant que i <  j
        alors début
        Placer (t , i , j , k)
        % on choisit le plus petit %
        si (j - k) > (k - i)
                alors début
                Quicksort (t , i , k - 1)
                i ← k + 1
                fin

                sinon début
```

```
                Quicksort (t , k + 1 , j)
                j ← k - 1
                fin
        fin

fin
```

Heapsort[3]

C'est un tri par *sélection* des minimums successifs basé sur une structure de *tas*. On va obtenir un tri *O (n log n)* comparaisons au pire, *sans mémoire auxiliaire*.

Arbres partiellement ordonnés

Un tri par sélection nécessite de savoir localiser et récupérer efficacement (en *O (1)*, si possible) le minimum parmi les éléments non encore placés.
On considère le cas particulier du type abstrait *ensemble* où les seules opérations sont :

- l'accès à un élément minimum
- la suppression du minimum
- l'adjonction d'un nouvel élément

On représente cette ensemble par *un arbre binaire parfait partiellement ordonné.*

Un arbre partiellement ordonné est un arbre étiqueté par les valeurs d'un ensemble muni d'*un ordre total*, tel que la valeur associée à tout nœud soit ≤ aux valeurs associées aux fils. La racine contient toujours un élément minimum, accès en *O (1)*.

1) Adjonction d'un élément x
On ajoute d'abord *x* comme une feuille en conservant la structure d'arbre binaire parfait. Puis, on reconstruit l'ordre partiel :

```
y ← x
tant que ( y ≠ racine ) et ( y < père(y) ) faire
        échanger y et père(y)
```

2) Suppression du min
Une fois la racine enlevée, on place la dernière feuille à la racine, pour conserver la structure d'arbre binaire parfait. Puis, on reconstruit l'ordre partiel :

```
y ← racine
tant que ( y n'est pas une feuille ) et ( y n'est pas ≤ aux deux fils ) faire
        échanger y et son fils de valeur minimum
```

La complexité en nombre de comparaisons de l'adjonction et de la suppression du minimum est au pire en *O (hauteur de l'arbre)*. Par ailleurs, la *hauteur* d'un arbre binaire parfait ayant *n* nœuds est de $\lfloor \log_2 n \rfloor$.

On utilise la représentation en tableau (*ordre hiérarchique*) des arbres binaires parfaits. (→ voire partie sur les structures arborescentes) Le tableau forme *le tas*.

On donne les algorithmes écrits en LDA des procédure d'adjonction et de suppression du minimum :

```
Procédure ajouter ( e/s t : tableau de 1 à N entiers , e/s n : entier , e/ x : entier )
% ajoute l'élément x à t ayant n éléments au moment de l'appel %
utilise localement i : entier
début
si n < N
        alors début
```

[3] Tri par tas

```
            n++
            t[n] ← x
            i ← n
            tant que ( i > 1 ) et ( t[i] < t[i div 2] ) faire
                        début
                        échanger t[i] et t[i div 2]
                        i ← i div 2
                        fin
            fin
sinon écrire débordement du tas
fin

Procédure suppress_min ( e/s t : tableau de 1 à N entiers , e/s n : entier , /s min : entier )
% fournit dans min le minimum pour t ayant n > 0 éléments %
utilise localement i, j : entiers
début
min ← t[1]
t[1] ← t[n]
n--
i ← 1
% tant que l'on est pas sur une feuille %
tant que ( i ≤ n div 2)  faire
            début
            si ( n = 2i ) ou ( t[2i] < t[2i+1])
                        alors j ← 2i
                        sinon  j ← 2i +1
            si ( t[i] > t[j])
                        alors début
                        échanger t[i] et t[j]
                        i ← j
                        fin
                        sinon exit
            fin
fin
```

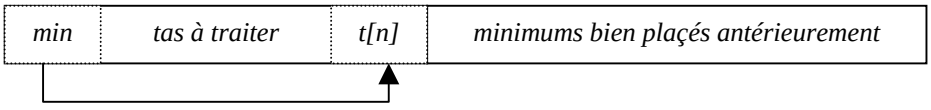
Tri par tas

Principe :

- Construire un tas contenant les *n* éléments par adjonction successives ; en *O (n log n).*
- Tant que le tas n'est pas vide, faire supprimer le minimum du tas avec réorganisation, mettre ce minimum à sa place définitive ; en *O (n log n).*

La complexité en comparaison est au pire en *O(n log n)*.

On utilise un seul tableau pour le tas et les valeurs des minimums successifs. Le minimum à récupérer est toujours dans *t[1]*. On mettra les minimums successifs à la droite du tableau, de la droite vers la gauche. A la fin, on obtient l'ensemble dans l'ordre décroissant.




```
Procédure heapsort (e/s t : tableau de 1 à n entiers)
utilise localement p, min : entiers
début
% construction du tas %
p ← 0
tant que p < n  faire
            ajouter( t , p , t[p+1] )
```

```
% tri %
tant que p > 1  faire
        suppress_min ( t , p , min )
        t[p+1] ← min
fin
```

Remarque : l'incrémentation et la décrémentation de p est généré par les procédures en e/s.