1 Basics

1.1 recherche séquentielle

```
trouvé = faux
i = 0
tant que (non trouvé) et (i < n)
   si T[i] = x, trouvé = vrai
   sinon i = i+1
si pas trouvé alors trouvé = faux
```

complexité : $O(n)$

1.2 recherche dichotomique

Si la liste triée, complexité en $O(\log n)$

1.3 tableau

C'est une collection de variables de même type.
Accès à chaque case en $O(1)$

1.4 liste chaînée

Accès au 1$^{\text{er}}$ maillon en $O(1)$
Accès au k-ième maillon en $O(n)$
insertion d'un élément en tête : en $O(1)$

```
add = nouvelle adresse
add.next = start
start = add
```

1.5 pile et file

	Tableau	Liste chaînée
Pile (LIFO) - Last In First Out	Ajout / suppression en tête (O(1))	Ajout / suppression en tête (O(1))
File (FIFO) - First In First Out	Ajout / suppression en O(1) variables début (incr. quand on supprime) et fin (incr. quend on ajoute)	Ajout / suppression en O(1). Il faut un pointeur début et un pointeur fin

2 Arbre

Structure composée de sommets regroupés de la façon suivante :
- une racine
- sous arbres disjoints (sous arbres). Il peut y en avoir 0 ou plus

hauteur (n nœuds)

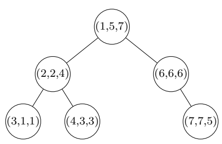
$$\left.\begin{array}{l} 0 \text{ si } n = 1, \\ 1 \text{ si } n > 1 \end{array}\right\} \leq h \leq n - 1.$$

2.1 Parcours d'un arbre $(R, A_1, A_2, \ldots, A_k)$

préfixe :
examiner R
pour i variant de 1 à k
    parcourir A_i en préfixe

postfixe :
pour i variant de 1 à k
    parcourir A_i en postfixe
examiner R

Arbre binaire



un AB est de la forme (Ar, Ag), Av avec A = vide ou AB, Arbre gauche et Ab, Arbre binaire.

parcours infixe :
parcourir A_1 en infixe
examiner R
parcourir A_2 en infixe

hauteur d'un AB (n nœuds) :
$s = n - 1$

Tri :
Thm : un tri comparatif a une complexité dans le pire des cas au moins $O(n \log n)$

Sélection :
principe : on cherche le plus petit élément et on le met au début

pseudo code :

```
pour i allant de 1 à n-1
   indicePetit = i
min = T[i]
   pour j allant de i+1 à n
       si T[j] < min
          indicePetit = j
          min = T[j]
   échanger (T[i], T[indicePetit])
```

complexité : $O(n^2)$

insertion :
principe : on suppose les i premiers éléments triés. On insère $T[i+1]$ en commençant par les remonter jusqu'à l'endroit où il doit aller

pseudo code :

```
pour i allant de 2 à n
   clé = T[i]
   j = i-1
   tant que (j >= 1) et (T[j] > clé)
       T[j+1] = T[j]
       j = j-1
   T[j+1] = clé
```

complexité : $O(n^2)$

Tri rapide :
principe : on utilise une fonction partition qui s'intéresse aux données entre p et d et forme la liste :
$\leq T[i] \mid T[i] > T[j]$

pseudo code (triRapide(p,d)) :

```
if p < d then
   q = partition(p,d)
   triRapide(p, q-1)
   triRapide(q+1, d)
```

$O(\log n)$ dans le pire des cas
$O(\log n)$ dans le meilleur des cas

Tri par Arbre Binaire de Recherche :
Un ABR est un AB dont les nœuds sont pourvus d'une clé.
La clé de tout nœud est comprise entre celles de ses fils. Les sous arbres à gauche sont plus petits et à droite plus grands.

La racine : on parcourt l'AB (on insère les données en fonction) dans l'ordre préfixe et on obtient un arbre équilibré.

insertion d'un élément :
Jusqu'à trouver la feuille : on va à gauche si la clé < racine clé, sinon à droite.
insertion en $O(h) = O(\log n)$ si l'arbre est équilibré.

Tri tas :
un arbre parfait : un AB est parfait si tous les niveaux sont remplis.
le tas est un AB complet (rempli)
la place des feuilles est de n/2
complexité : $O(n \log n)$

Représentation par un tableau :
N nœud d'indice i alors les fils de N ont indices 2i et 2i+1
le père de N d'indice i est i/2

Construction d'un tas :
insertion dans le tas donnée en tas à p reprises.
tant que la clé du fils est supérieure à celle du parent on permute.
$O(n \log n)$
construction en $O(n)$ $(n \log(n))$
On répète alors le procédé suivant :
on l'échange avec la racine et on le place en bas. Puis l'élément du haut redescend à sa place, tant qu'il est supérieur à un plus grand fils en échangeant.
Le coût de la remontée est $O(\log n)$
et on fait ça n fois donc $O(n \log n)$
Complexité : $O(n \log n)$

Hachage :
On veut vérifier l'appartenance de mots $x \in \{x_1, x_2, \ldots, x_N\}$ de liste L de m mots dans un alphabet Σ.
Principe de Hachage :
On utilise deux hachages h_1, h_2 :
pour $u \in \Sigma^*, h_1(u) = \ldots, h_2(u) = \ldots$
On redimensionne les hachages, si trop de collisions, on re-hash.
Si $n \gg m$ pas plus qu'on hache tous les codes pour E.
Il existe $\Phi(prob)$ pour savoir si $E \in L$ ou non.
Complexité : $O(n/m)$

gestion des collisions :
on crée un tableau de listes chaînées dont les maillons ont la même image par la hach. On utilise la liste chaînée en séquentielle.

complexité : $O(n/p)$
$O(n+p)$ moyenne $= O(p)$
$p = synhach$
longueur message : $O(n+p)$