Structures et fonctions supplémentaires :

- *void viderBuffer()*
  - Fonction permettant de vider le buffer de l'entrée standard
  - L'intérêt est d'éviter des problèmes lors de la saisie de l'utilisateur

- *void afficherStockLArbre(T_ListeVaccins* listeVaccins, int hauteur)*
  - Fonction permettant d'afficher la liste d'un nœud, la représentation graphique dans la console est conforme aux hauteurs respectives des nœuds.
  - Paramètres :
    - T_ListeVaccins* listeVaccins: liste de vaccins à afficher
    - int hauteur: hauteur du nœud en question

- *void afficherStockAArbre(T_ABR* abr, int hauteur)*
  - Fonction permettant d'afficher l'arbre de dates sous forme se rapprochant d'un arbre, afin de mieux visualiser la structure de ce dernier. La représentation graphique dans la console est conforme aux hauteurs respectives des nœuds.
  - Paramètres :
    - T_ABR* abr : l'arbre à afficher
    - int hauteur : la hauteur du noeud parcouru

- *int compterVaccinsListe(T_ListeVaccins* listeVaccins, char* marque)*
  - Fonction permettant de compter le nombre de vaccins disponibles dans une liste
  - Paramètres :
    - T_ListeVaccins* listeVaccins : la liste à parcourir
    - char* marque: le nom du vaccin recherché dans la liste
  - Retour : le nombre de vaccins de la marque recherchée

- *T_ABR** successeur(T_ABR* abr)*
  - Fonction permettant d'obtenir un pointeur vers le successeur d'un noeud de l'arbre
  - Paramètres :
    - T_ABR* abr : l'arbre dans lequel chercher le successeur
  - Retour : un pointeur vers le successeur

- *void supprimerNoeud(T_ABR** abr, char* date)*
  - Fonction permettant de supprimer un noeud de l'arbre en libérant son emplacement mémoire
  - Paramètres :
    - T_ABR** abr : un pointeur vers l'arbre

        - char* date: la date du noeud à supprimer

- *int comparerDates(char* date1, char* date2)*
    - Fonction permettant de comparer deux dates
    - Paramètres :
        - char* date1 : date1 à comparer avec date2
        - char* date2 : date2 à comparer avec date1
    - Retour : 1 si date1 > date2
      -1 si date1 < date2
      0 si date1 = date2

- *int verifValidDate(char* date)*
    - Fonction permettant de vérifier la conformité du format d'une date
    - Paramètres :
        - char* date : date à soumettre au test de validité
    - Retour : 1 si la date est valide
      0 sinon

- *void viderArbre (T_ABR** abr)*
    - Fonction supprimant tous les noeuds de l'arbre passé en paramètre
    - Paramètres :
        - T_ABR** abr : l'arbre à vider

Complexité des fonctions :

- *ajouterVaccinL*:
  Cas défavorable: insertion en queue de liste
  Pour une liste de taille N, on parcourt alors toute la liste avant d'insérer
  => Complexité en **O(N)**

- *ajouterVaccinA*:
  Cas défavorable: insertion dans le nœud en bout d'arbre, dans un arbre dégénéré, avec le vaccin inséré absent de la liste de vaccins de ce dernier nœud.
  Pour un arbre de hauteur H, on descend alors tout l'arbre avant d'insérer une nouvelle feuille. Dans le dernier nœud, celui où se fera l'insertion (donc de même date que le vaccin à insérer), on parcourra alors toute la liste de vaccins de taille N avant d'insérer le nouveau stock.
  => Complexité en **O(H+N)**

- *afficherStockL*: Pour une liste de taille N => Complexité en **O(N)**

- *afficherStockA*:
  Cas défavorable (pour une hauteur donnée) : l'arbre est équilibré et complet (tous les nœuds ont deux fils à l'exception des feuilles, qui sont toutes à la même hauteur.
  Pour un arbre de hauteur H, dont les nœuds comportent des listes de taille N, on parcourt chaque liste de chaque nœud de l'arbre, et le nombre de nœuds est égal à 2^(H+1)-1. On doit afficher chaque nœud de l'arbre. Pour chaque nœud, on affichera chaque vaccin de sa liste de taille N.
  => Complexité en O((2^(H+1)-1)*N) = **O(N*2^H)**

- *afficherStockLArbre*: Pour une liste de taille N, dont le nœud est de hauteur H dans l'arbre. Chaque élément est affiché avec un nombre de tabulations en fonction de H.
  => Complexité en **O(N*H)**

- *afficherStockAArbre*:
  On affiche l'arbre avec la même complexité qu'afficherStockA, soit O(N*2^H), en effectuant une boucle supplémentaire dans chaque appel récursif afin d'afficher les tabulations permettant de visualiser la différence de hauteur entre les nœuds. Cette boucle supplémentaire a toujours pour complexité O(H) avec H la hauteur du nœud parcouru.
  => Complexité en **O(H*N*2^H)**

- *compterVaccinsListe*:
  Cas défavorable: vaccin à compter se trouve en queue de liste
  Pour une liste de taille N, on parcourt alors toute la liste avant de le compter
  => Complexité en **O(N)**

- *compterVaccins*:
  Cas défavorable: l'arbre est équilibré et complet et chaque nœud comporte le vaccin en bout de liste.
  Pour un arbre de hauteur H, dont les nœuds comportent des listes de taille N, on parcourt chaque liste de chaque nœud de l'arbre, et le nombre de nœuds est égal à 2^(H+1)-1. On doit compter dans chaque liste le vaccin, complexité en O(N).
  Parcourir tous les nœuds de l'arbre est en complexité de O(2^(H+1)-1) = O(2^H).
  => Complexité en O((2^(H+1)-1)*N) = **O(N*2^H)**

- *deduireVaccinL*:
  Cas défavorable : vaccin à déduire se trouve en queue de liste
  Pour une liste de taille N, on parcourt alors toute la liste avant de le déduire
  => Complexité en **O(N)**

- *deduireVaccinA*:

Cas défavorable: l'arbre est équilibré et complet, chaque nœud comporte le vaccin en bout de liste, et le nombre de vaccin à déduire est égal au nombre total de vaccin en question, entraînant la suppression de nœuds.
Pour un arbre de hauteur H, dont les nœuds comportent des listes de taille N, on parcourt chaque liste de chaque nœud de l'arbre, et le nombre de nœuds est égal à 2^(H+1)-1. On doit déduire dans chaque liste le vaccin, complexité en O(N).
Parcourir tous les nœuds de l'arbre est en complexité de O(2^(H+1)-1) = O(2^H).
La suppression d'un nœud a pour complexité O(H).
=> Complexité en O((2^(H+1)-1)*N+H) = **O(N*2^H)**

- *successeur*:
  Cas défavorable: le nœud dont on cherche le successeur est la racine.
  On doit alors descendre tout l'arbre avant de trouver son successeur.
  => Complexité en **O(H)**

- *supprimerNoeud*:
  Cas défavorables: le nœud à supprimer est la racine (car il s'agit du cas défavorable de la recherche de successeur), ou le nœud à supprimer est une feuille (il faudra alors parcourir toute la hauteur pour le supprimer).
  Une fois le nœud atteint, on supprime toute la liste de ce nœud => O(N).
  => Complexité en **O(H+N)**

- *viderBuffer* : **O(1)**

- *comparerDates*: **O(1)**

- *verifValidDate*: **O(1)**

- *viderArbre*:
  Parcours postfixe de l'arbre passé en paramètres (pour une complexité en O(N)) et suppression de chaque nœud parcouru via supprimerNoeud. On remarquera que supprimerNoeud est appelé directement sur le nœud à supprimer : la complexité de cet appel sera alors en O(1).
  => Complexité en **O(N)**