Arbres Binaires
Première implémentation - v3

Ce document a pour objectif de vous familiariser avec l'implémentation des arbres binaires. Vous devrez y utiliser une pile et une file à un moment donné pour les algorithmes itératifs appliqués aux arbres.

Pour rappel, les arbres binaires sont constitués de *nœuds* stockant une *clé* (l'élément ou l'identifiant de l'élément), et chaque nœud dispose de liens vers un *fils gauche* et un *fils droit*.

1 Types, structures, et exemples

Toutes les clés qui seront stockées seront strictement supérieures à 0.

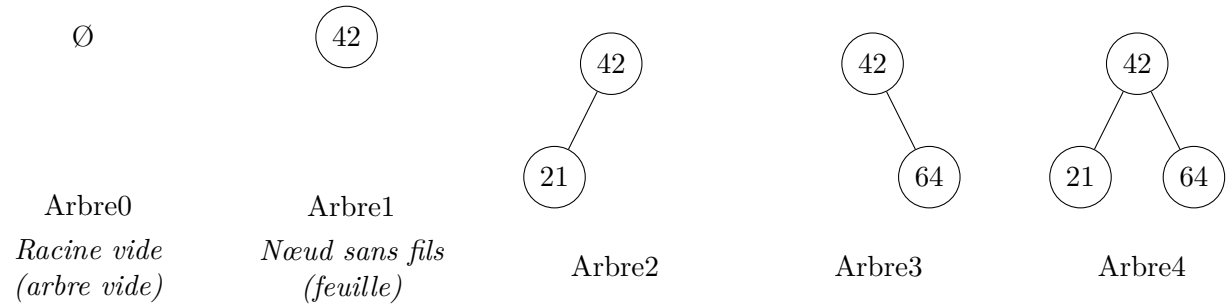
1) Implémentez une structure d'arbre binaire nommée **`node`** pouvant contenir une clé qui sera un entier. L'arbre doit être stocké avec des pointeurs entre des nœuds.
2) Implémentez maintenant une structure d'arbre binaire nommée **`bin_tree_t`** qui stockera un arbre binaire sous forme de tableau (n'oubliez pas qu'un tableau a besoin que l'on stocke sa taille dans un entier). Les nœuds vides contiendront -1.
3) Implémentez une fonction **`create_bin_tree_t`** créant un **`bin_tree_t`** dont tous les nœuds seront à -1 (un arbre vide). La fonction prendra en paramètre un entier servant de taille maximale au tableau.

```
bin_tree_t *create_bin_tree_t(int max_len);
```

4) Implémentez maintenant *à la main* les arbres suivants. Pour cela vous allez écrire une fonction renvoyant un pointeur vers un **`node`** racine ou un **`bin_tree_t`**. (La version *node* des arbres doit être entièrement faite à la main, mais vous êtes libres pour la version *bin_tree_t*)

```
node *build_Arbre0_p(void);
node *build_Arbre1_p(void);
node *build_Arbre2_p(void);
node *build_Arbre3_p(void);
node *build_Arbre4_p(void);
```

```
bin_tree_t *build_Arbre0_t(void);
bin_tree_t *build_Arbre1_t(void);
bin_tree_t *build_Arbre2_t(void);
bin_tree_t *build_Arbre3_t(void);
bin_tree_t *build_Arbre4_t(void);
```



Arbre0 — *Racine vide (arbre vide)*

Arbre1 — *Nœud sans fils (feuille)*

Arbre2

Arbre3

Arbre4

5) Implémentez une fonction **`create_node`** créant un **`node`**. Cette fonction prendra un entier en paramètre qui sera mis dans la clé, et les adresses des deux fils.

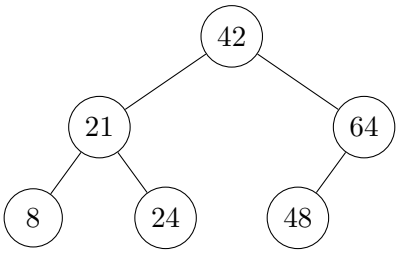
```
node *create_node(int key, node *left_child, node *right_child);
```

6) Implémentez *à la main* avec les fonctions *create_node* et *create_bin_tree_t* les arbres suivants, toujours en produisant des fonctions générant des arbres et renvoyant un pointeur vers le nœud racine ou vers la structure contenant le tableau.

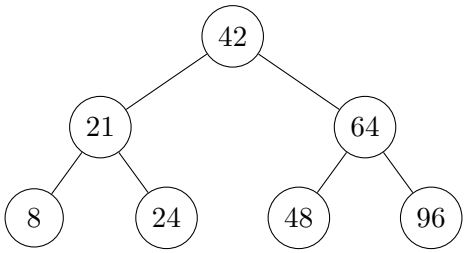
Astuces : allouez d'abord chaque nœud grâce à une fonction dédiée, écrivez une ligne pour placer chaque nœud comme fils d'un autre, et commencez par l'arbre le plus rempli pour pouvoir copier/coller le code et retirer les nœuds qui ne servent pas.

```
node *build_Arbre5_p(void);
node *build_Arbre6_p(void);
node *build_Arbre7_p(void);
node *build_Arbre8_p(void);
node *build_Arbre9_p(void);
node *build_Arbre10_p(void);
node *build_Arbre11_p(void);
node *build_Arbre12_p(void);
```

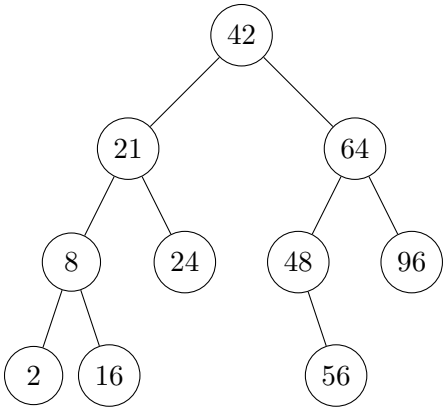
```
bin_tree_t *build_Arbre5_t(void);
bin_tree_t *build_Arbre6_t(void);
bin_tree_t *build_Arbre7_t(void);
bin_tree_t *build_Arbre8_t(void);
bin_tree_t *build_Arbre9_t(void);
bin_tree_t *build_Arbre10_t(void);
bin_tree_t *build_Arbre11_t(void);
bin_tree_t *build_Arbre12_t(void);
```



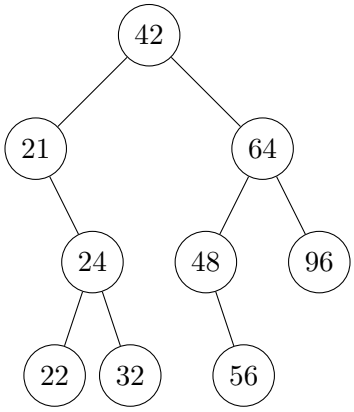
Arbre5



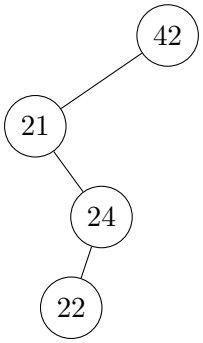
Arbre6



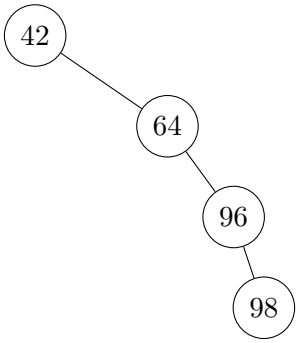
Arbre7



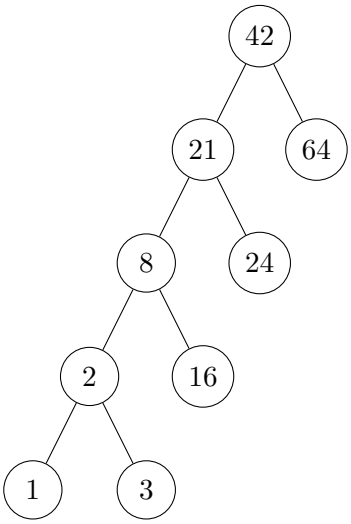
Arbre8



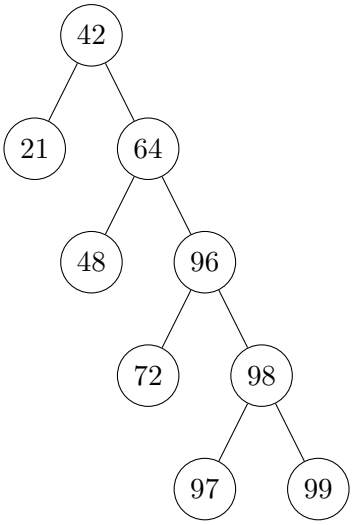
Arbre9



Arbre10



Arbre11



Arbre12

Le but de ces exemples est de vous aider à tester les algorithmes suivants. Utilisez donc ces exemples pour vérifier le bon fonctionnement des algorithmes suivants.

2 Premiers algorithmes (récursifs)

Vous allez maintenant implémenter les fonctions de base analysant et parcourant des arbres. Ces fonctions fonctionnent en autonomie (elles n'ont pas besoin de pile ou de file externes), mais certaines peuvent être des fonctions chapeau (donc ayant besoin de sous fonctions).

7) Implémentez les fonctions récursives suivantes calculant la taille et la hauteur de l'arbre. (Pour rappel : la *taille* est le nombre de nœuds contenus dans l'arbre, et la *hauteur* est le niveau du nœud le plus profond de l'arbre, sachant que la racine est à la hauteur 0)

```
int size_btp_r(node *T);
int height_btp_r(node *T);
```

8) Implémentez maintenant les mêmes fonctions pour un arbre stocké sous forme de tableau (la récursion n'est pas obligatoire).

```
int size_btt(bin_tree_t *T);
int height_btt(bin_tree_t *T);
```

9) Implémentez trois fonctions récursives effectuant un parcours profondeur main gauche, et affichant les clés des nœuds dans l'ordre préfixe, infixe, ou suffixe.

```
int print_dfs_preorder_btp_r(node *T);
int print_dfs_inorder_btp_r(node *T);
int print_dfs_postorder_btp_r(node *T);
```

10) Implémentez maintenant une fonction écrivant le numéro d'ordre hiérarchique de chaque nœud suivi de la clé contenue.

```
int print_hierarchical_btp_r(node *T);
```

11) Implémentez maintenant une fonction renvoyant le numéro d'ordre hiérarchique du nœud contenant la clé données en paramètre. Si la clé n'est pas trouvée, alors on renverra -1.

```
int hierarchical_number_btp_r(node *T, int key);
```

12) Implémentez une fonction construisant un arbre binaire sous forme de tableau à partir d'un arbre binaire sous forme de pointeurs.

Pour limiter les réallocations, vous avez le droit d'appeler une fois au tout début la fonction *size* que vous avez précédemment réalisée.

```
bin_tree_t *btp_r_to_btt(node *T);
```

13) Implémentez une fonction transformant un arbre binaire sous forme de pointeurs en un arbre binaire sous forme de pointeurs *avec* un champs supplémentaire dans la structure des nœuds : la taille du sous-arbre démarrant au nœud courant.

Il est nécessaire de définir une nouvelle structure **`node_ext`** dans laquelle vous ajouterez un champs supplémentaire.

```
node_ext *extend_btp_r(node *T);
```

3 Algorithmes itératifs

Pour cette seconde partie, il est nécessaire de disposer d'une file et d'une pile fonctionnelles. C'est-à-dire qu'elles doivent implémenter au *minimum absolu* ces fonctions (pour permettre une correction automatique, vous devez strictement respecter ces prototypes) :

```
/* Selon l'implementation, max_size peut ne pas etre pris en compte */
my_stack *create_stack(int sizeof_elt, int max_size);

void *head_stack(my_stack *s);
my_stack *push(my_stack *s, void *elt);
my_stack *pop(my_stack *s);

void delete_stack(my_stack *s);
```

```
/* Selon l'implementation, max_size peut ne pas etre pris en compte */
my_queue *create_queue(int sizeof_elt, int max_size);

void *head_queue(my_queue *q);
my_queue *enqueue(my_queue *q, void *elt);
my_queue *dequeue(my_queue *q);

void delete_queue(my_queue *q);
```

Comme vous l'aurez remarqué, au lieu de prendre un élément sous forme d'entier, ces structures prendront des pointeurs vers ces éléments. La taille de chaque élément est indiqué lors de la création de la structure par le paramètre *sizeof_elt* (la structure contiendra donc N éléments de taille *sizeof_elt*). La suppression d'un élément (avec *dequeue* ou *pop*) ne doit pas libérer de la mémoire ces éléments. Le paramètre *max_size* donné à la fonction *create_stack* doit absolument exister, mais peut ne pas être utilisé si vous implémentez les piles/files avec des pointeurs (le code doit néanmoins compiler avec tous les flags de compilation : vous devez utiliser l'astuce pour forcer les variables à exister).

14) Implémentez une fonction itérative effectuant un parcours largeur dans l'ordre hiérarchique et affichant la clé de chaque nœud.

```
int print_bfs_btp_i(node *T);
```

15) Implémentez trois fonctions itératives effectuant un parcours profondeur main gauche, et affichant les clés des nœuds dans l'ordre préfixe, infixe, ou suffixe.

```
int print_dfs_preorder_btp_i(node *T);
int print_dfs_inorder_btp_i(node *T);
int print_dfs_postorder_btp_i(node *T);
```

Ce document et ses illustrations ont été réalisés par Fabrice BOISSIER en mars 2023. Certains exercices sont inspirés des supports de cours de Nathalie "Junior" BOUQUET, et Christophe "Krisboul" BOULLAY.