# Arbres Binaires
**Problèmes**

Ce document a pour objectif de vous confronter à des problèmes impliquant des arbres binaires. Vous devrez avoir compris tous les concepts basiques de l'algorithmique pour les résoudre : la récursivité, les tableaux, les pointeurs, ainsi que les structures de données précédentes (listes, piles, files, arbres binaires).

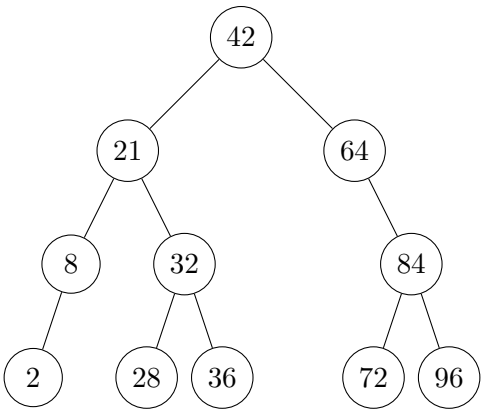
Pour rappel, les arbres binaires sont constitués de *nœuds* stockant une *clé* (l'élément ou l'identifiant de l'élément), et chaque nœud dispose de liens vers un *fils gauche* et un *fils droit*.

*Dans l'ensemble des exercices, toutes les clés qui seront stockées seront strictement supérieures à 0.*

## Algorithmes

### Conversion Arbre Binaire -> Tableau

Le but de l'exercice est de convertir un arbre binaire au format **`node*`** vers un tableau contenant les clés. Pour chaque nœud vide, la case associée doit être remplie d'un **`-1`**. Voici un exemple d'arbre, ainsi que le tableau de sortie associé :




| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 42 | 21 | 64 | 8 | 32 | *-1* | 84 | 2 | *-1* | 28 | 36 | *-1* | *-1* | 72 | 96 |
| *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* | *10* | *11* | *12* | *13* | *14* | *15* |

Pour effectuer cet algorithme, plusieurs solutions sont possibles, mais toutes s'appuient sur le numéro hiérarchique. La plus simple consiste à effectuer un parcours profondeur, mais il est envisageable de faire un parcours largeur à la place. N'oubliez pas que *toutes* les cases équivalentes à un nœud vide doivent contenir **`-1`**.

1) Implémentez une fonction *node_to_tab* prenant en paramètre la racine d'une arbre binaire au format **`node*`**, ainsi qu'un tableau de **`node*`** déjà alloué avec suffisamment d'espace (et sa taille) :

```
void node_to_tab(node *root, node **tab, int size);
```

# 1 Problème A

Dans ce problème, vous allez devoir construire le *plus court chemin* entre 2 nœuds quelconques d'un arbre. Un *chemin* est simplement une liste de nœuds reliés les uns après les autres par des liens. Dans le cas des arbres, la racine est reliée à ses fils, et vice-versa. Il existe par exemple une infinité de chemins entre D et C dans l'arbre suivant, mais un seul plus court chemin :

— Plus court chemin : D - B - A - C

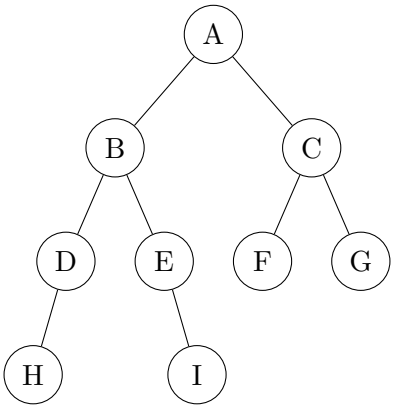
— Exemple 1 de chemin : D - B - E - B - A - C

— Exemple 2 de chemin : D - B - D - B - A - C - F - C

Pour résoudre ce problème, vous devrez écrire des fonctions et procédures en C. Aucune fonction de la bibliothèque C ne pourra être utilisée exceptées *malloc*, *free*, et *printf* : vous devez réécrire toute fonction utile, sauf si une consigne précise le contraire. La macro *sizeof* est autorisée.

## 1.1 Questions préalables

### 1.1.1 Indiquez le plus court chemin entre les nœuds suivants :




| | | | | | | |
|---|---|---|---|---|---|---|
| B → I | | | | | | |
| A → B | | | | | | |
| B → A | | | | | | |
| A → I | | | | | | |

| | | | | | | |
|---|---|---|---|---|---|---|
| D → E | | | | | | |
| A → D | | | | | | |
| D → A | | | | | | |
| A → E | | | | | | |

| | | | | | | |
|---|---|---|---|---|---|---|
| I → F | | | | | | |
| A → I | | | | | | |
| I → A | | | | | | |
| A → F | | | | | | |

### 1.1.2 De quel algorithme récursif peut-on dériver la recherche d'un chemin dans un arbre entre la racine et un nœud ?

## 1.2 Résolution générale du problème

Dans les exercices suivants, nous considèrerons les arbres binaires comme cette structure **node** :

```
typedef struct node
{
  int          key;
  struct node *lc;  // Left child
  struct node *rc;  // Right child
} node;
```

En admettant que vous disposez des fonctions suivantes exclusivement pour cette question, écrivez l'algorithme général de résolution du problème sous la forme d'une fonction C. (Les tableaux de node* sont tous *NULL-terminated*).

*Rappel : un tableau NULL-terminated est un tableau dont la dernière case contient la valeur NULL.*

Un tableau NULL-terminated contenant A, B, et C sera de la forme suivante : | A | B | C | *NULL* |

```
// Longueur du chemin (nombre de noeuds) entre la racine et un noeud
//      (si le chemin n'existe pas, la fonction renvoie -1)
int count_path_root_to_node(node *root, node *end);

// Generation du tableau (NULL-terminated) de node* contenant le
//   chemin entre la racine et un noeud
node **build_path_root_to_node(node *root, node *end);

// Longueur d'un tableau de node*
int array_length(node **in_array);

// Inversion en place d'un tableau de node*
//   (en place = sans le reallouer)
void invert_array_in_place(node **in_array);

// Longueur du prefixe commun entre 2 tableaux
int common_prefix_length(node **tab1, node **tab2);

// Fusion de 2 tableaux de node* vers un nouveau tableau
node **merge_arrays_new(node **tab1, node **tab2);
```

Vous pouvez également appeler les fonctions classiques des arbres : *hauteur(arbre)*, *profondeur(nœud)*, *taille(arbre)*, *parc_prof(arbre)*, ... mais vous devrez les réimplémenter si nécessaire dans les questions suivantes.

### 1.2.1 Écrivez la fonction produisant un tableau *NULL-terminated* du plus court chemin entre deux nœuds quelconques d'un arbre binaire

```
node **build_path(node *root, node *start, node *end)
```

## 1.3 Génération des tableaux de chemins

### 1.3.1 Écrivez une fonction comptant le nombre de nœuds sur le chemin entre la racine d'un arbre et un nœud. Si aucun chemin n'existe vers ce nœud, vous devez renvoyer -1 :

*Exemple : entre le nœud B et I du schéma illustratif en page 3, il y a 3 nœuds : B - E - I*

```
int count_path_root_to_nodes(node *root, node *end)
```

### 1.3.2 Écrivez une fonction générant un tableau *NULL-terminated* de nœuds allant de la racine d'un arbre à un nœud :

```
node **build_path_root_to_node(node *root, node *end)
```

## 1.4 Gestion des tableaux

### 1.4.1 Écrivez une fonction calculant la taille d'un tableau *NULL-terminated* de nœuds :

Exemple : Un tableau contenant | A | B | C | *NULL* | sera considéré comme de taille 3

```
int array_length(node **in_array)
```

### 1.4.2 Écrivez une procédure inversant l'ordre des éléments d'un tableau *NULL-terminated* contenant des node*. Si le tableau est un pointeur NULL, renvoyez NULL. L'inversion doit se faire en place, c'est-à-dire qu'il ne faut pas allouer de nouveau tableau. :

Exemple : Un tableau contenant | A | B | C | *NULL* | sera inversé en | C | B | A | *NULL* |

```
void invert_array_in_place(node **in_array)
```

### 1.4.3 Écrivez une fonction calculant la longueur du préfixe commun de deux tableaux *NULL-terminated* :

Exemple : Un tableau contenant | A | B | C | *NULL* | et un contenant | A | B | F | *NULL* | ont un préfixe commun de longueur 2

```
int common_prefix_length(node **tab1, node **tab2)
```

### 1.4.4 Écrivez une fonction fusionnant deux tableaux *NULL-terminated* en produisant un nouveau tableau :

Exemple : Un tableau contenant | A | B | *NULL* | et un contenant | D | E | *NULL* | produiront en sortie le tableau | A | B | D | E | *NULL* |

```
node **merge_arras_new(node **tab1, node **tab2)
```

*Ce document et ses illustrations ont été réalisés par Fabrice BOISSIER en mai 2025. Certains exercices sont inspirés des supports de cours de Nathalie "Junior" BOUQUET, et Christophe "Krisboul" BOULLAY.*