[CYBER1][2023-2024] CORRECTION Examen (2h00)

Algorithmique 2

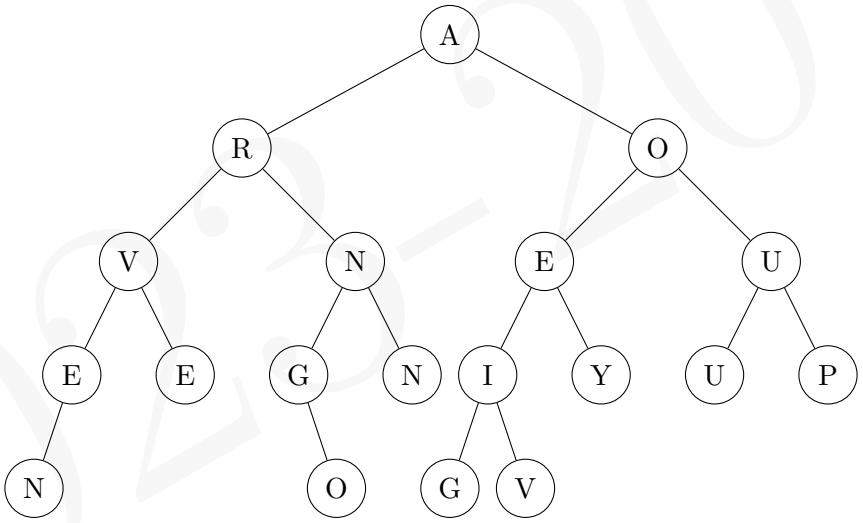
NOM : PRÉNOM :

Vous devez respecter les consignes suivantes, sous peine de 0 :

I) Lisez le sujet en entier avec attention
II) Répondez sur le sujet
III) Ne détachez pas les agrafes du sujet
IV) Écrivez lisiblement vos réponses (si nécessaire en majuscules)
V) Ne trichez pas

1 Arbres Binaires (10 points)

1.1 (3 points) Indiquez toutes les propriétés que possède cet arbre, puis écrivez les clés lors d'un parcours profondeur main gauche de l'arbre dans les 3 ordres ainsi que lors d'un parcours largeur :



Arité : 2 Taille : 19 Hauteur : 4 Nb feuilles : 9

☐ Arbre binaire strict / localement complet
☐ Arbre binaire (presque) complet
☐ Arbre binaire parfait
☐ Arbre filiforme
☐ Peigne gauche
☐ Peigne droit

Parcours profondeur :

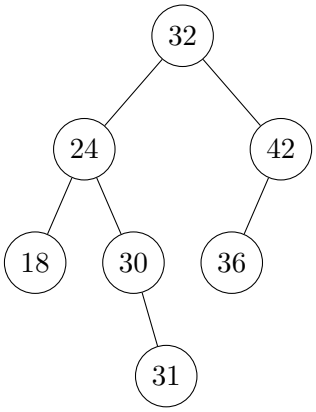
ordre préfixe :	A	R	V	E	N	E	N	G	O	N	O	E	I	G	V	Y	U	U	P
ordre infixe :	N	E	V	E	R	G	O	N	N	A	G	I	V	E	Y	O	U	U	P
ordre suffixe :	N	E	E	V	O	G	N	N	R	G	V	I	Y	E	U	P	U	O	A

Parcours largeur :

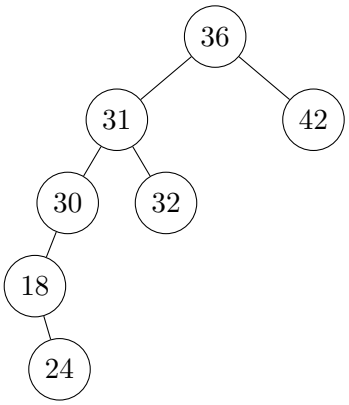
ordre infixe :	A	R	O	V	N	E	U	E	E	G	N	I	Y	U	P	N	O	G	V

1.2 (5 points) Dessinez le résultat de l'insertion dans cet ordre précis des éléments suivants dans un ABR (insertion en feuille et en racine) et dans un AVL :

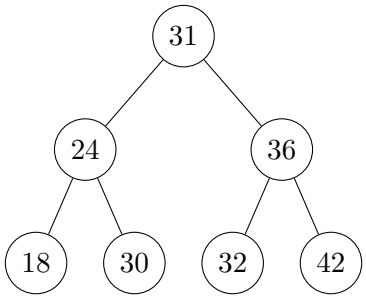
Éléments insérés : 32 - 24 - 18 - 30 - 31 - 42 - 36



ABR
insertion en feuille
(1 point)



ABR
insertion en racine
(2 points)



AVL
(2 points)

1.3 (2 points) Écrivez une fonction récursive « *parc_prof_rec* » effectuant un parcours profondeur main gauche dans un arbre binaire, et affichant les nœuds dans chacun des ordres :

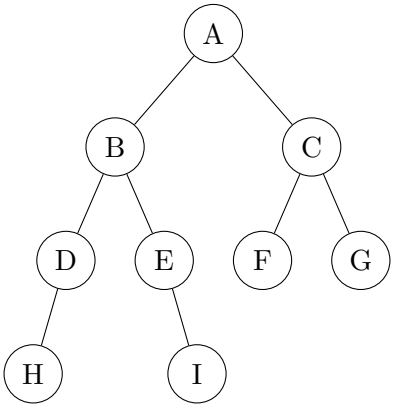
Il faut expliciter les éventuels ordres au format : « Ordre : nœud » (exemple : « Préfixe : 42 »)

2 Micro Projet (10 points)

Afin de tester l'ensemble des compétences acquises au cours de cette année, vous allez maintenant toutes les exploiter pour traiter un problème simple. La question s'intéresse au nombre et à la façon de construire les chemins entre deux nœuds d'un même arbre. Vous devez mobiliser absolument toutes les connaissances vues en cours pour pouvoir réussir cet exercice.
Vous devez écrire les fonctions et procédures suivantes en C. Aucune fonction de la bibliothèque C ne pourra être utilisée exceptées *malloc* et *free* : vous devez réécrire toute fonction utile, sauf si une consigne précise le contraire.

Un chemin est 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.

Par exemple, dans le schéma suivant, il existe un chemin entre B et I constitué de : B - E - I



2.1 Questions préalables [2 points]

2.1.1 (1,5 point) Indiquez les chemins entre les nœuds suivants :

B → I	B	E	I			
A → B	A	B				
B → A	B	A				
A → I	A	B	E	I		

D → E	D	B	E			
A → D	A	B	D			
D → A	D	B	A			
A → E	A	B	E			

I → F	I	E	B	A	C	F
A → I	A	B	E	I		
I → A	I	E	B	A		
A → F	A	C	F			

2.1.2 (0,5 point) Combien de chemin(s) existe-t-il au maximum entre deux nœuds quelconques d'un arbre ?

1 : Il n'existe qu'un seul chemin entre deux nœuds quelconques d'un arbre.

2.2 Résolution générale du problème [2 points]

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
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_len(node **in_array);

// Inversion en place d'un tableau de node*
//   (en place = sans le reallouer)
node **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.

2.2.1 (2 points) Écrivez la fonction produisant un tableau *NULL-terminated* du chemin entre deux nœuds quelconques d'un arbre binaire

```
node **build_path(node *root, node *start, node *end)
{
  node **root_path1, **root_path2;
  node **path1, **path2;
  node **final_path;
  int prefix_end;

  root_path1 = build_path_root_to_node(root, start);
  root_path2 = build_path_root_to_node(root, end);

  prefix_end = common_prefix_next_index(root_path1, root_path2);
  path1 = &root_path1[prefix_end];
  path2 = &root_path2[prefix_end - 1];

  path1 = invert_array_in_place(path1);

  final_path = merge_arrays_copy(path1, path2);

  free(root_path1)
  free(root_path2);

  return (final_path);
}
```

2.3 Génération des tableaux de chemins [2 points]

2.3.1 (2 points) 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_nodes_rec(node *root, node *end, int level)
{
  int left, right;

  if (root == NULL)
    return (-1);

  if (root == end)
    return (level);

  left = count_path_nodes_rec(root->lc, end, level + 1);
  right = count_path_nodes_rec(root->rc, end, level + 1);

  if (left > right)
    return (left);
  else
    return (right);
}
```

```
int count_path_nodes(node *start, node *end)
{
  return (count_path_nodes_rec(start, end, 1));
}
```

2.4 Gestion des tableaux [4 points]

2.4.1 (2 points) Écrivez une fonction inversant l'ordre des éléments d'un tableau *NULL-terminated* contenant des node*, et renvoyant l'adresse de la nouvelle 1ère case. 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* |.

```
node **invert_array_in_place(node **in_tab)
{
  node *tmp;
  int max = 0;
  int i = 0;

  if (in_tab == NULL)
    return (NULL);

  while (in_tab[max] != NULL)
    max++;

  while (i < max)
  {
    tmp = in_tab[i];
    in_tab[i] = in_tab[max];
    in_tab[max] = tmp;
    i++;
    max--;
  }
  return (in_tab);
}
```

2.4.2 (2 points) Écrivez une fonction prenant en paramètres deux tableaux *NULL-terminated* de node* et les fusionnant en un seul tableau *NULL-terminated*. Le tableau de sortie doit être un nouveau tableau alloué. Si les tableaux sont des pointeurs NULL, renvoyez NULL :

```
int tab_len(node *tab)
{
  int i = 0;

  if (tab == NULL)
    return (0);

  while (tab[i] != NULL)
    i++;
  return (i);
}
```

```
node **merge_arrays(node **tab1, node **tab2)
{
  node **out = NULL;
  int len1, len2, i = 0;

  if ((tab1 == NULL) && (tab2 == NULL))
    return (NULL);

  len1 = tab_len(tab1);
  len2 = tab_len(tab2);
  out = malloc((len1 + len2 + 1) * sizeof (node*));
  while (i < len1)
  {
    out[i] = tab1[i];
    i++;
  }

  i = 0;
  while (i < len2)
  {
    out[len1 + i] = tab2[i];
    i++;
  }

  out[len1 + len2] = NULL;

  return (out);
}
```