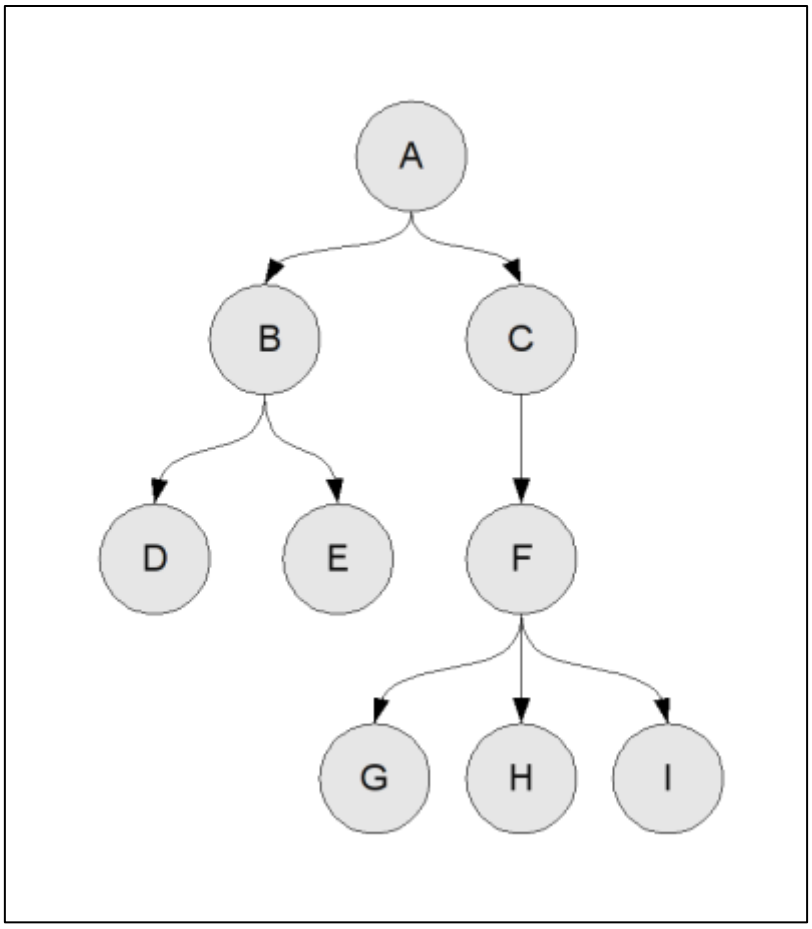
ARBRES BINAIRE ET N-AIRE

Travail de Fiona ISS, Romain SCHLOTTER, Valentin ZELLER, Kévin LI et Olivier FRANÇOIS
A Ludus-Académie, 2019-2020

I) Définition – Qu'est-ce qu'un arbre ?

En informatique, un arbre est une structure semblable à une liste. Cette structure est très utilisée aussi bien dans le jeu-vidéo (Path-Finding d'IA) ou plus simplement dans des algorithmes concernant les bases de données, ou l'arborescence de fichiers. On peut le représenter de la façon suivante :



Chaque élément de l'arbre possède donc un certain nombre de frères (ou sibling) et d'enfants (ou child). On verra plus tard que selon le type d'arbre, ce nombre est fixé.

II) Vocabulaire

Un arbre est composé de plusieurs éléments :

- Les feuilles : éléments ne possédant pas de fils dans l'arbre. Sur la représentation ci-dessus, D, E, G, H et I sont des feuilles.
- Les nœuds : éléments possédant des fils. Sur la représentation ci-dessus, B, C, et F sont des nœuds.
- Une racine : élément ne possédant pas de parent. Sur la représentation ci-dessus, A est la racine.

En plus de ces éléments, ces éléments possèdent des caractéristiques :

- La profondeur d'un nœud : se dit de la "distance" séparant le nœud de la racine de l'arbre. Par exemple, la profondeur du nœud F est 1 (ou 2 si on compte le nœud en lui-même).
- La hauteur d'un arbre : se dit de la plus grande profondeur d'une feuille de l'arbre. Par exemple, la hauteur de l'arbre illustré ci-dessus est 3 (A jusqu'à G, H ou I)
- La taille d'un arbre : se dit du nombre de nœud (éventuellement sans les feuilles) de l'arbre. Par exemple, sur l'illustration ci-dessus, l'arbre a une taille de 8.
- La longueur de cheminement : somme de la profondeur de chacune des feuilles. Sur l'illustration ci-dessus, la longueur de cheminement vaut 13.
- L'étiquette d'un nœud : désigne la donnée que stock le nœud.
- Le degré d'un nœud : désigne le nombre d'enfant que possède un nœud.

III) Les types d'arbres

On distingue plusieurs types d'arbres selon le nombre d'enfants/frères qu'on décide que chaque nœud possède.

A) Les arbres binaires

Les arbres binaires sont des arbres où chaque élément possède au plus 2 enfants. On peut ainsi les modéliser de la façon suivante :

```
typedef struct BinaryTree
{
    int value;
    struct BinaryTree *left;
    struct BinaryTree *right;
    struct BinaryTree *parent;
}BinaryTree;
```

Il est à noter que certains modèles ne prennent pas la peine de garder en mémoire pour chaque nœud son parent. Cependant, il est essentiel de modéliser les deux enfants, que l'on appelle souvent enfant gauche et enfant droit.
La structure contient donc des pointeurs vers ces éléments ainsi qu'un entier : c'est l'étiquette du nœud. Cette étiquette pourrait cependant être de n'importe quel type : chaine, structure…

Nous allons donc désormais nous intéresser aux méthodes permettant de manipuler ces arbres.

1) Création

```c
BinaryTree *CreateBinaryTree(int value)
//Fonction qui prend en paramètre une valeur (la valeur d'un noeud, ici un int) et créer un arbre constitué d'un seul noeud
{
    BinaryTree *tree = malloc(sizeof(*tree));

    if (tree == NULL)
        exit(EXIT_FAILURE);

    tree->value = value;
    tree->left = NULL;
    tree->right = NULL;
    tree->parent = NULL;

    return tree;
}
```

Cette méthode permet de créer un arbre simple d'un seul nœud, avec aucun enfant et aucun parent.

2) Insertion, suppression

Cette fonction d'insertion va joindre une racine à deux enfants, créant les liaisons nécessaires (parent à enfant). On peut la combiner à la création pour créer et joindre directement des nœuds.

```c
BinaryTree *JoinTreeFromExistingRoot(BinaryTree *left, BinaryTree *right, BinaryTree *root)
//Fonction qui va permettre de joindre 2 arbre à un 3ème
//On passe en paramètre l'arbre gauche à joindre, l'arbre droite à joindre, et la racine à laquelle on va joindre les deux arbres
{

    //on assigne les pointeurs correspondant aux arbres
    root->left = left;
    root->right = right;
    //on assigne les parents
    if (left != NULL)
        left->parent = root;

    if (right != NULL)
        right->parent = root;
    //et on retourne l'arbre
    return root;
}
```

Cette fonction va prendre en paramètre la référence d'un nœud et la libérer, ainsi que ses enfants. Elle est appelée récursivement pour tout libérer. Ainsi on peut l'appeler sur un parent, un enfant, selon ce que l'on souhaite libérer en mémoire.

```c
void ClearBinaryTree(BinaryTree *tree)
//Fonction qui va free la mémoire allouée à L'arbre passé en paramètre, La fonction va être appelée de manière récursive pour free tous les noeuds de l'arbre
{
    //si on arrive sur un pointeur NULL, alors c'est que le noeud en question n'existe pas. du coup on return et on remonte on noeud d'au-dessus
    if (tree == NULL)
        return;
    //sinon on appelle la fonction de manière récursive sur L'arbre gauche de L'arbre en question
    if (tree->left!=NULL)
        ClearBinaryTree(tree->left);
    //puis le droit
    if (tree->right!=NULL)
        ClearBinaryTree(tree->right);
    //et du coup on free L'arbre

    free(tree);
}
```

3) Taille, profondeur

Cette fonction va retourner la taille de l'arbre (le nombre de nœuds) en comptant la racine.

```c
int GetBinaryTreeSize(BinaryTree *tree)
//Renvoie le nombre de noeuds actuel dans L'arbre passé en paramètre, racine comprise
{
    if (tree == NULL)
        return 0;

    return (GetBinaryTreeSize(tree->left) + GetBinaryTreeSize(tree->right) + 1);
}
```

4) Recherche, tri

Pour la recherche, on a cette fonction qui va retourner l'équivalent d'un booléen (1 si on a trouvé la valeur, 0 si non)

```c
//retourne 1 si on a trouvé la valeur, 0 sinon
int SearchBinaryTree(BinaryTree *tree, int value)
{
    //booleen qui va déterminer si on a trouvé la valeur ou non
    int hasBeenFound = 0;

    //si on a trouvé la valeur on place le booleen à 1
    if (tree->value == value)
        hasBeenFound = 1;
    //sinon on regarde l'enfant de gauche en premier (s'il n'est pas NULL) et si on n'a pas encore trouvé le chiffre
    if (tree->left != NULL && !hasBeenFound)
        hasBeenFound = SearchBinaryTree(tree->left,value);
    //pareil à droite
    if (tree->right != NULL && !hasBeenFound)
        hasBeenFound = SearchBinaryTree(tree->right,value);


    //inévitablement si on l'a trouvé on va retourner 1 vu que les tests ne seront plus effectués, sinon 0 sera retourné
    return hasBeenFound;
}
```

Concernant le tri, il y a différentes méthodes. Il y a ce qu'on appelle le « heap-sort » ou le « tri par tas ».
On part d'un tableau de valeurs, on le transforme en tas (arbre binaire où tous les niveaux de l'arbre sont remplis, à part éventuellement le dernier niveau de l'arbre, où les valeurs seront remplies en priorité à gauche puis à droite).

A partir de là, on va créer un « max-heap » ou « tas-max », où chaque nœud parent à une valeur supérieur à ses nœuds enfants. A partir de là, on inverse le premier nœud et le dernier nœud, et on supprime le dernier nœud, qui est notre plus grande valeur. On répète ainsi la création de tas-max, d'inversion/suppression des valeurs, jusqu'à qu'il n'y ait plus qu'une seule valeur. La résultante est notre tableau trié en valeurs croissantes.

```c
//cette fonction va créer un arbre binaire avec le tableau de valeur passé en paramètre
//elle prend i en paramètre car dans le tableau on peut déterminer qu'à un index i, les enfants sont situés à 2*i+1 et 2*i+2
BinaryTree *CreateBinaryTreeFromArray(int arr[], BinaryTree* root, int i, int arraySize)
{
    if (i < arraySize)
    {
        //on va créer un arbre avec la valeur arr[i]
        BinaryTree *temp = CreateBinaryTree(arr[i]);
        root = temp;
        //et on appelle récursivement la fonction sur la gauche puis sur la droite
        root->left = CreateBinaryTreeFromArray(arr,root->left,2*i+1,arraySize);
        root->right = CreateBinaryTreeFromArray(arr,root->right,2*i+2,arraySize);
        //et on lie le parent et ses enfants ensemble
        JoinTreeFromExistingRoot(root->left,root->right,root);
    }
    return root;
}
```

Création de l'arbre binaire depuis un tableau de valeurs

```c
//fonction qui va créer un tas-max sur l'arbre passé en paramètre
//on passe également le tableau de valeurs afin d'interchanger les valeurs dans le tableau pendant que l'on créer le tas-max
void CreateMaxHeap(BinaryTree *tree, int arr[], int arraySize)
{
    if (tree == NULL)
        return;
    //on considère que la première valeur est la plus grande au début
    int largest = tree->value;

    //si l'enfant de gauche n'est pas NULL donc il existe
    if (tree->left != NULL)
    {
        //si sa valeur est plus grande, alors on remplace les valeurs dans l'arbre, et on les interchange dans le tableau
        if(tree->left->value > largest)
        {
            int tempValueToReplace1 = SearchIndexOfValue(arr, largest, arraySize);
            int tempValueToReplace2 = SearchIndexOfValue(arr, tree->left->value, arraySize);
            tree->value = tree->left->value;
            tree->left->value = largest;
            largest = tree->value;
            InvertPlaceInArray(arr, tempValueToReplace1, tempValueToReplace2);
        }
    }
    //pareil pour la droite
    if (tree->right != NULL)
    {
        if(tree->right->value > largest)
        {
            int tempValueToReplace1 = SearchIndexOfValue(arr, largest, arraySize);
            int tempValueToReplace2 = SearchIndexOfValue(arr, tree->right->value, arraySize);
            tree->value = tree->right->value;
            tree->right->value = largest;
            largest = tree->value;
            InvertPlaceInArray(arr, tempValueToReplace1, tempValueToReplace2);
        }
    }

    //et on répercute sur les enfants de gauche et droite
    if(tree->left != NULL)
        CreateMaxHeap(tree->left, arr, arraySize);
    if(tree->right != NULL)
        CreateMaxHeap(tree->right, arr, arraySize);
}
```

Création du tas-max (ne fais qu'une seule rotation, il faudra l'appeler dans une boucle afin de récupérer la valeur maximale tout en haut de l'arbre)

```c
//Cette fonction va inverser le premier et le dernier noeud, et supprimer le dernier noeud
void InvertFirstAndLastNodeAndRemoveLastNode(BinaryTree *binaryTree, int arr[], int arraySize)
{
    //on récupère le premier noeud (forcément c'est celui passé en paramètre)
    BinaryTree *firstNode = binaryTree;
    //on utilise notre fonction pour récupérer le dernier noeud
    BinaryTree *lastNode = GetLastNode(binaryTree, arr[arraySize-1]);

    //on inverse les valeurs dans l'arbre binaire
    int tempValue = firstNode->value;
    firstNode->value = lastNode->value;
    lastNode->value = tempValue;

    //on inverse les places dans le tableau
    InvertPlaceInArray(arr,0, arraySize-1);


    //on récupère la référence du parent, et on supprime la référence sur l'enfant qui va être supprimer
    BinaryTree *previousNode = lastNode->parent;
    if(previousNode->left->value == lastNode->value)
        previousNode->left = NULL;
    else
        previousNode->right = NULL;

    //et on free la valeur supprimée
    ClearBinaryTree(lastNode);
```

Inversion des premiers et derniers nœuds, et suppression du dernier nœud à la fin

```c
//cette fonction inverse 2 valeurs dans un tableau
void InvertPlaceInArray(int arr[], int indexToInvert1, int indexToInvert2)
{
    int temp = arr[indexToInvert1];
    arr[indexToInvert1] = arr[indexToInvert2];
    arr[indexToInvert2] = temp;
}
```

On pourra retrouver une fonction qui permet d'inverser les valeurs dans notre tableau de valeur également

```c
//cette fonction permet de retourner le dernier node (qui est la dernière valeur du tableau)
BinaryTree *GetLastNode(BinaryTree *binaryTree, int value)
{
    //on créer un arbre temporaire
    BinaryTree *tempTree = NULL;

    //si on le trouve, on lui passe la référence du noeud
    if (binaryTree->value == value)
        tempTree = binaryTree;

    //on appelle récursivement la fonction tant que l'on a pas trouvé le noeud
    if (binaryTree->left != NULL && tempTree == NULL)
        tempTree = GetLastNode(binaryTree->left,value);

    if (binaryTree->right != NULL && tempTree == NULL)
        tempTree = GetLastNode(binaryTree->right,value);


    //on retourne le dernier noeud
    return tempTree;
}
```

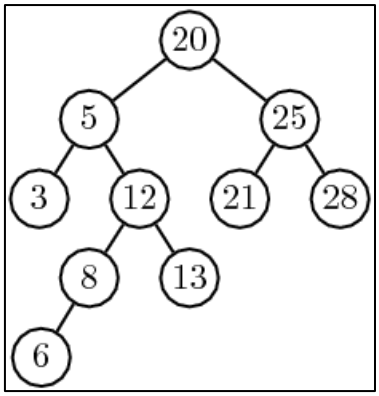
Cette fonction va retourner le dernier nœud qui correspond à la dernière valeur du tableau

5) L'arbre binaire de recherche.

La structure de l'arbre binaire peut être équilibrée lors de sa création afin de permettre la recherche et le tri de ses éléments.

Un arbre binaire se construit de la manière suivante :

- Nous avons une liste de données à renseigner dans cet arbre que nous souhaitons classer, prenons des entiers.
- Le premier élément de la liste devient la racine de l'arbre.
- Puis chaque élément dont la valeur est inférieure est placé en tant qu'enfant à gauche du nœud actuel, de même chaque élément dont la valeur est supérieure ou égale est placé en tant qu'enfant à droite du nœud actuel.
- Si le nœud enfant choisi est déjà occupé, on reprend l'opération précédent avec ce nouvel élément en tant que nœud actuel.



```
17  Noeud* InsertionArbreBinRecherche(Noeud* pRacine, int nValeur)
18  //BUT : Insérer une valeur dans un arbre binaire de recherche.
19  //ENTREE : Une valeur et la racine de l'arbre binaire de recherche.
20  //SORTIE : L'arbre binaire mis à jour avec la valeur en plus placé au bon endroit.
21  {
22      if (pRacine == NULL) //Si le noeud est vide
23      {
24          //On crée le noeud.
25          pRacine=CreationNoeud(nValeur);
26      }
27      else //Si l'arbre n'est pas vide on faire le test.
28      {
29          if (nValeur < pRacine->nValeur)
30          {
31              //Si la valeur est inférieure à la valeur actuel on va à gauche.
32              pRacine->aGauche = InsertionArbreBinRecherche(pRacine->aGauche,nValeur);
33          }
34          else
35          {
36              //Si la valeur est supérieure à la valeur actuel on va à droite.
37              pRacine->aDroit = InsertionArbreBinRecherche(pRacine->aDroit,nValeur);
38          }
39      }
40      return pRacine;
41  }
```

Code pour l'insertion des données dans un arbre binaire de recherche

```
43  int RechercheArbreBin(Noeud* pRacine, int nValeur)
44  //BUT : Savoir si l'arbre contient cette valeur ou non.
45  //ENTREE : Une valeur et la racine de l'arbre binaire de recherche.
46  //SORTIE : 1 si l'arbre contient la valeur et 0 sinon.
47  {
48      if(pRacine==NULL)
49      {
50          //Si on tombe sur un noeud vide c'est qu'on n'a pas trouvé la valeur cherchée.
51          return 0;
52      }
53      else if (pRacine->nValeur==nValeur)
54      {
55          //Si on a trouvé la valeur on le renvoie.
56          return 1;
57      }
58      else if (nValeur < pRacine->nValeur)
59      {
60          //Si la valeur est inférieure à la valeur actuel on va à gauche.
61          return RechercheArbreBin(pRacine->aGauche,nValeur);
62      }
63      else
64      {
65          //Si la valeur est supérieure à la valeur actuel on va à droite.
66          return RechercheArbreBin(pRacine->aDroit,nValeur);
67      }
68  }
```

Code pour la recherche de l'existence d'une donnée dans un arbre binaire de recherche

```
86  void RenvoieTriArbreBin(Noeud *pRacine, int* tTable,int* nIndexArbreBinaireRecherche)
87  //BUT : Mettre à jour un tableau à partir des données de l'arbre binaire.
88  //ENTREE : Le tableau et l'arbre binaire qui a été créé à partir de ce tableau.
89  //SORTIE : le tableau mis à jour et classé grâce à l'arbre binaire de recherche.
90  {
91      if (pRacine==NULL)
92          return;
93
94      RenvoieTriArbreBin(pRacine->aGauche,tTable, nIndexArbreBinaireRecherche);
95
96      tTable[*nIndexArbreBinaireRecherche]=pRacine->nValeur;
97      (*nIndexArbreBinaireRecherche)++;
98
99      RenvoieTriArbreBin(pRacine->aDroit,tTable, nIndexArbreBinaireRecherche);
100 }
```

Code pour renvoyer le tableau trié avec l'utilisation d'un arbre binaire de recherche correspondant

B) Les arbres n-aires

Les arbres n-aires sont une généralisation des arbres binaires. Dans ces arbres-ci, chaque nœud peut avoir au plus n enfants. Puisque n peut être aussi grand que l'on veut, il n'est pas possible pour un nœud, comme pour les binaires, de le faire pointer vers ses n-enfants. Dans ce modèle-ci, on préfère donc faire pointer un nœud vers

- Son premier enfant (child)
- Son premier frère (sibling)

Cela peut ainsi donner quelque chose comme ça :

```c
typedef struct Tree
{
    int value;
    struct Tree *sibling;
    struct Tree *child;
}Tree;
```

Tout comme pour les arbres binaires, voici quelques méthodes pour manipuler ces arbres.

1) Création

```c
Tree *CreateTree(char value, Tree *child, Tree *sibling)
//Creer un arbre a  partir des parametres passes.
{
    Tree *tempTree = malloc(sizeof(*tempTree));

    if (tempTree == NULL)
    {
        printf("ERREUR DANS L'ALLOCATION : l'arbre n'a pas pu etre cree.\n");
        return NULL;
    }

    tempTree->value = value;
    tempTree->child = child;
    tempTree->sibling = sibling;

    return tempTree;
}
```

2) Insertion, suppression

```c
void InsertFirstChild(Tree *parent, Tree *newChild)
{
    if (parent == NULL)
        return NULL;

    //Si le parent a déjà un child, il devient le sibling du nouveau child
    if (parent->child != NULL)
        newChild->sibling = parent->child;

    parent->child = newChild;
}
```

Pas grand-chose à dire de plus sur l'insertion.

La suppression est un peu plus délicate, il y a deux plus deux types de suppressions : celle où l'on supprime un enfant et celle où l'on supprime un sibling. Les deux étant néanmoins assez proche, nous n'allons ici expliciter que la suppression d'un enfant. Il faut penser à tous les cas ; pour cela prenons l'arbre suivant :

Imaginons que l'on souhaite supprimer l'enfant de A : le nœud B.

- Il faut commencer par déclarer D, l'enfant de B, comme étant l'enfant de A
- Puis déclarer C, le sibling de B, comme étant le sibling de E.

Ainsi, pour chaque suppression il faut donc penser à vérifier :

- Est-ce que le nœud à supprimer à un child ?
    - Si oui, cet enfant devient le nouveau child du parent.
- Est-ce que le nœud à supprimer à un sibling ?
    - Si oui, ce sibling devient le sibling du dernier child du nœud à supprimer.

Le code étant plutôt long, le lecteur est invité à le consulter dans le cbp associé à cette documentation.

3) Taille, hauteur

Les fonctions de taille et de hauteur étant semblables, nous ne parlerons ici que de la fonction calculant la taille.

```
int GetTreeSize(Tree *node)
//Renvoie la taille (nombre de noeuds) d'un arbre a  partir du noeud donne en comptant la racine.
{
    if (node == NULL)
    {
        ChangeColor(color_Red, color_Black);
        printf("ERREUR ARBRE VIDE\n");
        ChangeColor(color_White, color_Black);

        return -1;
    }

    int count = 1;

    Tree *currentNode = node->child;
    while (currentNode != NULL)
    {
        count += GetTreeSize(currentNode);
        currentNode = currentNode->sibling;
    }

    return count;
}
```

La fonction se base sur le principe de récursivité. On commence par partir de la racine de l'arbre puis l'on regarde son premier child. Si ce child possède lui-même un child, on le regarde (jusqu'à arriver à

un nœud sans child) tout en incrémentant un compteur. Puis on fait pareil sur le sibling et lorsque l'on atteint un nœud n'ayant ni sibling ni child, on renvoie le compteur et on remonte au premier parent ayant un sibling (grâce à la récursivité) puis on recommence.

4) Binarisation d'un arbre n-aire