ENSTA Paris
2023-2024

Résolution de problèmes algorithmiques – IN103

Danil Berrah et Alexandre Chapoutot

Feuille d'exercices 3

Objectif(s)

- Manipuler une bibliothèque d'arbres binaires ;
- Manipuler une bibliothèque d'arbres binaires de recherche ;
- Découvrir les algorithmes d'apprentissage automatique fondés sur les arbres de décisions.

PRÉPARATION

Cette première partie permet de préparer votre environnement (**si vous ne l'avez pas fait au dernier TD**) de travail afin de pouvoir utiliser facilement la bibliothèque logicielle **libin103** spécialement développée pour cet enseignement.

1. A la racine de votre compte, créez un répertoire nommé `Library` s'il n'a pas déjà été créé, puis placez vous dans ce répertoire.

   ```
   mkdir ~/Library; cd ~/Library
   ```

2. Téléchargez l'archive `libin103-1.4.tar.gz` sur le site du cours

   ```
   wget https://perso.ensta-paris.fr/~chapoutot/teaching/in103/practical-work/libin103-1.4.tar.gz
   ```

3. Désarchivez l'archive

   ```
   tar -xvzf libin103-1.4.tar.gz
   ```

4. Allez dans le répertoire `libin103-1.4` et compilez la bibliothèque. Quelle commande faut-il utiliser ?
   — Il faut utiliser la commande `make`. A la fin de la compilation vérifier la présence du fichier `libin103.a` dans le répertoire `source`.
   — Également, vous pouvez exécuter la commande `make check` pour compiler et exécuter les programmes de tests.

EXERCICES

Exercice 1 – Prise en main de la bibliothèque des arbres binaires

L'objectif de cet exercice est de prendre en main la partie de la bibliothèque qui concerne les arbres binaires. Pour cela nous allons construire pas à pas un arbre binaire dont l'information contenue dans les nœuds sera des entiers.

Fonctions utiles pour cet exercice

- `integer_bitree_init`
- `integer_bitree_destroy`
- `integer_bitree_ins_right`
- `integer_bitree_ins_left`
- `integer_bitree_size`
- `integer_bitree_root`
- `integer_inorder`
- `integer_preorder`
- `integer_postorder`
- `integer_list_init`
- `integer_list_head`
- `integer_list_tail`
- `integer_list_next`
- `integer_list_data`
- `integer_list_destroy`

Question 1

Dans quels fichiers peut-on trouver le prototype de ces fonctions ?

Solution:

Dans le répertoire `include` de la bibliothèque `libin103`, plus précisément

- `integer_bitree.h`, pour se simplifier la tâche il est possible d'inclure `bitree.h` ;
- `integer_list.h`, pour se simplifier la tâche il est possible d'inclure `list.h` ;
- `integer_bitreealg.h`, pour se simplifier la tâche il est possible d'inclure `bitreealg.h`.

Explication(s):

- ✿ Nous aurons bien entendu besoin des fonctions de manipulations d'arbres binaires à valeurs entières.
- ✿ Nous allons utiliser les fonctions de parcours d'arbres.
- ✿ Nous aurons également besoin des listes pour effectuer un parcours d'arbre et afficher l'ordre du parcours.

Question 2

Créez une variable, nommée `tree`, dont le type est un arbre binaire d'entiers et initialisez cette variable.

Solution:

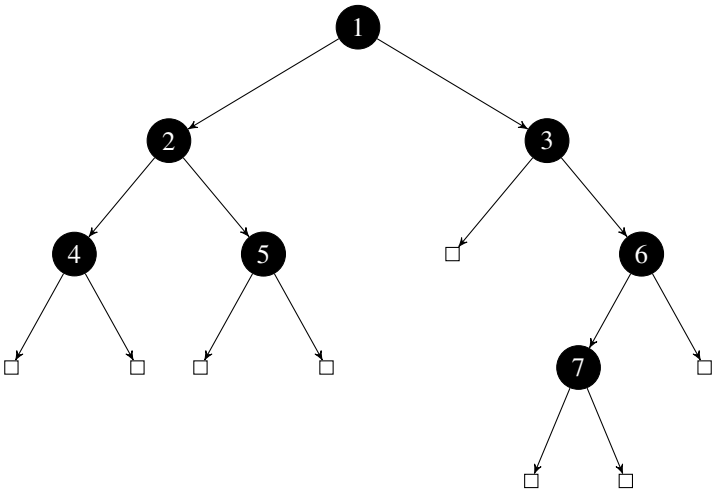
```
integer_bitree_t tree;
integer_bitree_init (&tree);
```

Explication(s):

- ✿ Attention au passage par adresse !
- ✿ D'ailleurs pourquoi on fait un passage par adresse ? => cela évite la recopie de la `struct` lors de l'appel aux fonctions de la bibliothèque.

Question 3

Construisez l'arbre binaire qui a la structure suivante



Solution:

```
/* Particularite de la bibliotheque pour l'insertion dans la racine */
integer_bitree_ins_left (&tree, NULL, 1);
printf ("Root takes value 1\n");

integer_bitreenode_t* root = integer_bitree_root (&tree);
integer_bitree_ins_left (&tree, root, 2);
printf ("Root->left takes value 2\n");
integer_bitree_ins_right (&tree, root, 3);
printf ("Root->right takes value 3\n");

integer_bitreenode_t* left = integer_bitree_left(root);
integer_bitree_ins_left (&tree, left, 4);
printf ("Root->left->left takes value 4\n");
integer_bitree_ins_right (&tree, left, 5);
printf ("Root->left->right takes value 5\n");

integer_bitreenode_t* right = integer_bitree_right(root);
integer_bitree_ins_right (&tree, right, 6);
printf ("Root->right->right takes value 6\n");

integer_bitreenode_t* rightright = integer_bitree_right(right);
integer_bitree_ins_left (&tree, rightright, 7);
printf ("Root->right->right->left takes value 7\n");
```

Explication(s):

- ✿ La fonction `integer_bitree_ins_left` ajoute une information à la racine de l'arbre quand le pointeur du second argument est `NULL`. Il faut donc penser à une phase d'initialisation dans l'insertion dans l'arbre pour gérer le cas particulier de l'insertion à la racine.
- ✿ Exceptez le cas particulier de la racine, il suffit de garder les pointeurs des différents nœuds de l'arbre pour ajouter petit à petit de nouveaux nœuds.

Question 4

Nous voulons maintenant faire un parcours de l'arbre, dans l'ordre infixe.

— Quelle fonction faut-il utiliser pour cela ?
— Quel fichier faut-il inclure ?

Solution:

— Il faut utiliser la fonction `integer_inorder` qui est déclarée dans le fichier `integer_bitreealg.h` qui est inclus dans l'interface générale `bitreealg.h`

Question 5

Mettez en œuvre le parcours infixe de l'arbre depuis la racine.

Solution:

```
integer_list_t t_inorder;
integer_list_init (&t_inorder);
root = integer_bitree_root (&tree);
integer_inorder (root, &t_inorder);
```

Explication(s):

- ✿ Le parcours de l'arbre va stocker les informations contenues dans les nœuds dans une liste, il faut donc déclarer au préalable une liste d'entiers et l'initialiser avec la fonction `integer_list_init`.
- ✿ Il faut également récupérer le pointeur sur la racine grâce à la fonction `integer_bitree_root`.

Question 6

Développez une fonction nommée, `print_integer_list` qui permet d'afficher le contenu d'une liste et dont le prototype est :

```
void print_integer_list (integer_list_t* list);
```

Utilisez cette fonction pour afficher l'ordre des nœuds de l'arbre dans l'ordre infixe.

Solution:
Le code de la fonction est

```
void print_integer_list (integer_list_t* list) {
  integer_list_elmt_t *elt;
  for (elt = integer_list_head(list);
       elt != integer_list_tail(list);
       elt = integer_list_next (elt)) {
    printf ("%d, ", integer_list_data(elt));
  }
  printf ("%d\n", integer_list_data(integer_list_tail(list)));
}
```

Explication(s):

- ✿ On retrouve ici le schéma classique d'une itération sur les listes pour la fonction `print_integer_list`.

Solution:

```
  printf ("List (inorder): ");
  print_integer_list (&t_inorder);
```

Explication(s):

- ✿ L'affichage des nœuds dans l'ordre du parcours infixe est simplifiée par l'utilisation de la fonction `print_integer_list`.

Question 7

Affichez l'ordre des nœuds de l'arbre suivant un parcours préfixe et postfixe.

Solution:

```
  integer_list_t t_preorder;
  integer_list_init (&t_preorder);

  integer_list_t t_postorder;
  integer_list_init (&t_postorder);

  integer_preorder (root, &t_preorder);
  integer_postorder (root, &t_postorder);

  printf ("List (preorder): ");
  print_integer_list (&t_preorder);

  printf ("List (postorder): ");
  print_integer_list (&t_postorder);
```

Explication(s):

- ✿ Les différents parcours d'arbre suivent le même schéma d'utilisation en remplissant une liste auxiliaire.
- ✿ On définit ainsi une liste pour le parcours préfixe et autre pour le parcours postfixe.

Question 8

Quelles fonctions faut-il appeler pour libérer la mémoire associée aux différentes structures de données utilisées ?

Solution:

```
  integer_bitree_destroy (&tree);
  integer_list_destroy (&t_preorder);
  integer_list_destroy (&t_inorder);
  integer_list_destroy (&t_postorder);
```

Explication(s):

- ✿ Il faut libérer la mémoire associée à l'arbre binaire et aux listes utilisées pour les parcours d'arbres en utilisant les fonctions de destruction associées.

Solution:La solution complète est :

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#include "bitree.h"
#include "bitreealg.h"

void print_integer_list (integer_list_t* list) {
  integer_list_elmt_t *elt;
  for (elt = integer_list_head(list);
       elt != integer_list_tail(list);
       elt = integer_list_next (elt)) {
    printf ("%d, ", integer_list_data(elt));
  }
  printf ("%d\n", integer_list_data(integer_list_tail(list)));
}

int main () {
  integer_bitree_t tree;
  integer_bitree_init (&tree);

  /* Particularite de la bibliotheque pour l'insertion dans la racine */
  integer_bitree_ins_left (&tree, NULL, 1);
  printf ("Root takes value 1\n");

  integer_bitreenode_t* root = integer_bitree_root (&tree);
  integer_bitree_ins_left (&tree, root, 2);
  printf ("Root->left takes value 2\n");
  integer_bitree_ins_right (&tree, root, 3);
  printf ("Root->right takes value 3\n");

  integer_bitreenode_t* left = integer_bitree_left(root);
  integer_bitree_ins_left (&tree, left, 4);
  printf ("Root->left->left takes value 4\n");
  integer_bitree_ins_right (&tree, left, 5);
  printf ("Root->left->right takes value 5\n");

  integer_bitreenode_t* right = integer_bitree_right(root);
  integer_bitree_ins_right (&tree, right, 6);
  printf ("Root->right->right takes value 6\n");

  integer_bitreenode_t* rightright = integer_bitree_right(right);
  integer_bitree_ins_left (&tree, rightright, 7);
  printf ("Root->right->right->left takes value 7\n");

  integer_list_t t_inorder;
  integer_list_init (&t_inorder);
  root = integer_bitree_root (&tree);
  integer_inorder (root, &t_inorder);

  printf ("List (inorder): ");
  print_integer_list (&t_inorder);

  integer_list_t t_preorder;
  integer_list_init (&t_preorder);

  integer_list_t t_postorder;
  integer_list_init (&t_postorder);

  integer_preorder (root, &t_preorder);
  integer_postorder (root, &t_postorder);

  printf ("List (preorder): ");
  print_integer_list (&t_preorder);

  printf ("List (postorder): ");
  print_integer_list (&t_postorder);

  integer_bitree_destroy (&tree);
  integer_list_destroy (&t_preorder);
  integer_list_destroy (&t_inorder);
  integer_list_destroy (&t_postorder);

  return EXIT_SUCCESS;
}
```

Exercice 2 – Un algorithme de tri avec des arbres binaires de recherche

Un arbre binaire de recherche permet de stocker des données en fonctions d'une relation d'ordre sur les valeurs. Nous allons considérer dans que des arbres avec valeurs entières.

Pour cet exercice, nous allons comparer la complexité d'un tri à partir d'un arbre binaire de recherche et un arbre binaire de recherche équilibré.

Fonctions utiles pour cet exercice

- `integer_bistree_init`
- `integer_bistree_destroy`
- `integer_bistree_insert`
- `generic_bitree_root`
- `generic_bitree_data`
- `generic_inorder`
- `generic_list_init`
- `generic_list_head`
- `generic_list_tail`
- `generic_list_next`
- `generic_list_data`
- `generic_list_destroy`
- `integer_list_init`
- `integer_list_head`
- `integer_list_tail`
- `integer_list_next`
- `integer_list_data`
- `integer_list_destroy`

Question 1

Rappelez la façon dont sont stockées les valeurs dans un arbre binaire de recherche (équilibré ou non).

Solution:
On note que dans l'implémentation des arbres binaires de recherche dans la bibliothèque `libin103`, les doublons ne sont pas autorisés donc cela va imposer une contraintes sur les données d'entrée à trier (pas de multi-occurence).
La propriété suivie par les arbres binaires de recherche est
— Enfant gauche plus petit que la valeur du nœud parent.
— Enfant droit plus grand que la valeur du nœud parent.

Question 2

Quel type de parcours d'arbre peut être utilisé pour suivre l'ordre croissant des valeurs dans un arbre binaire de recherche ?

Solution:
Explication(s):
✿ Le parcours infixe permet de traiter les informations dans l'ordre fils gauche, nœud, fils droit donc de conserver l'ordre croissant des valeurs.

Question 3

(Version simple) Dans cette question, nous allons considérer la structure de données AVL des arbres binaires de recherche équilibrée fournie avec la bibliothèque `libin103` (`bistree`). Mettez en œuvre un algorithme de tri en vous appuyant sur cette structure de données.

Solution:

```
integer_bistree_t avl;
integer_bistree_init (&avl);

for (int i = 0; i < size; i++) {
  integer_bistree_insert (&avl, tab[i]);
  printf ("insert␣%d␣=>␣root␣is␣%d\n", tab[i],
          ((integer_avlnode_t*)generic_bitree_data(generic_bitree_root(&avl)))->data);
}

generic_list_t sorted_avl;
generic_list_init (&sorted_avl, NULL, NULL, NULL);
generic_bitreenode_t* root_avl = generic_bitree_root (&avl);
generic_inorder (root_avl, &sorted_avl);
```

Explication(s):

- ✿ La principale difficulté dans la mise en œuvre de cet algorithme est la nécessité de faire un parcours de l'arbre avec une version générique (`void*`). En conséquence, on utilise également une liste générique pour stocker la séquence de valeur triée.
- ✿ Plus précisément, la mise en œuvre des AVL dans la bibliothèque `libin103` s'appuie sur les arbres binaires génériques (`generic_bitree`) dont le contenu est une structure de la forme

```
typedef struct integer_avlnode_t {
  int data;
  int hidden;
  int factor;
} integer_avlnode_t;
```

  Le champs `data` est la valeur entière à stocker, le champs `factor` permet d'indiquer le déséquilibre possible de l'arbre et le champs `hidden` est utilisé pour marquer le nœuds qui sont supprimés (car l'approche *lazy removal* est suivi dans l'implémentation des AVL).

- ✿ **Remarque :** effet de bord, les valeurs avec multi-occurence dans l'ensemble à trier seront conservées en un seul exemplaire.

Solution:
Fonction d'affichage sur pour le résultat du parcours infixe des AVL.

```
void print_avlnode_list (generic_list_t *list) {
  generic_list_elmt_t* elt = generic_list_head(list);
  for (; elt != generic_list_tail(list); elt = generic_list_next(elt)) {
    printf ("%d,␣", ((integer_avlnode_t*)generic_list_data(elt))->data);
  }
  printf ("%d\n", ((integer_avlnode_t*)generic_list_data(generic_list_tail(list)))->data);
}
```

Explication(s):

- ✿ Essentiellement il faut penser à transtyper (faire un cast) des valeurs stockées dans liste chaînée générique pour passer tu type `void*` au type `integer_avlnode_t`.
- ✿ A noter que nous devons exposer la structure du type `integer_avlnode_t` pour récupérer la valeur du champs `data`. Il n'y a pas de fonction accesseur pour ce type d'où l'utilisation de l'opérateur `->`.
- ✿ Autrement, nous avons la même structure d'itérateur sur les listes que nous avons déjà vu plusieurs fois.

Question 4

Discutez de la complexité de cet algorithme.

Solution:
On considère plusieurs étapes : le parcours de la liste de données, l'insertion dans l'arbre et le parcours d'arbre.
Explication(s):

- si on considère une complexité suivant le nombre n d'éléments manipulés, il faut n insertions dans l'arbre ;
- La fonction `insert` dans l'arbre binaire de recherche équilibré a une complexité de $\mathcal{O}(\log(n))$. On considère les rotations avec un coût constant.
- Le parcours infixe de l'arbre étant exhaustif on aura également une complexité linéaire $\mathcal{O}(n)$;
- Au final, on a un algorithme dans la classe $\mathcal{O}(n\log(n)+n)$.

Question 5

Mettez en œuvre une fonction qui permet de calculer la hauteur maximale d'un arbre dont la racine est donnée en argument. Quel style de programmation allez-vous utiliser ?

Solution:

```
int max_height (generic_bitreenode_t *root) {
  int hL = 0;
  int hR = 0;
  if (generic_bitree_is_eob(root)) {
    return 0;
  }

  if (generic_bitree_left (root) != NULL) {
    hL = max_height (generic_bitree_left (root)) + 1;
  }

  if (generic_bitree_right (root) != NULL) {
    hR = max_height (generic_bitree_right (root)) + 1;
  }

  return hL > hR ? hL : hR;
}
```

Explication(s):

- On utilise un style de programmation récursif pour exploiter la structure récursive du type de données arbres binaires.
- **Remarque** On considère un arbre générique binaire car on pourra utiliser cette fonction autant pour les AVL que pour d'autres arbres binaires. En particulier, seule la structure de l'arbre nous intéresse pas le contenu des nœuds !

Question 6

Nous allons maintenant mettre en œuvre un arbre binaire de recherche non équilibré. Définissez une fonction dont le prototype est

```
void add_node(integer_bitree_t* tree, integer_bitreenode_t* current, int elem);
```

Cette fonction permet d'ajouter un élément dans un arbre suivant la propriété des arbres binaires de recherche.

Avant de commencer le développement, quel style de programmation allez-vous utilisé ?

Solution:

```
void add_node(integer_bitree_t* tree, integer_bitreenode_t* current, int elem){

  int current_data = integer_bitree_data(current);

  if (elem != current_data) {
    if (elem <= current_data){
      if (integer_bitree_left (current) == NULL) {
        integer_bitree_ins_left (tree, current, elem);
      }
      else {
        add_node (tree, integer_bitree_left(current), elem);
      }
    } else {
      if (integer_bitree_right (current) == NULL) {
        integer_bitree_ins_right (tree, current, elem);
      } else {
        add_node (tree, integer_bitree_right(current), elem);
      }
    }
  }
}
```

Explication(s):

- ✿ Nous utilisons un style récursif qui s'appuie sur la structure récursive d'un arbre binaire.
- ✿ Nous ferons attention de ne pas ajouter de doublons dans l'arbre comme pour le cas des AVL.
- ✿ L'algorithme est intuitif. Si la valeur à ajouter est plus petite que la valeur du nœud on suit le fils gauche. Si la valeur à ajouter est plus grande que celle du nœud on suit le fils droit. On crée un nouveau nœud quand le fils gauche ou droit est `NULL`.

Question 7

Triez le tableau donné dans la fonction `main` avec les deux approches et comparer la hauteur des arbres résultats.

Solution:

— Avec le tableau donné en exemple, la hauteur de l'AVL est 3 tandis que la hauteur de l'arbre non équilibré est 5.

— On peut essayer avec un tableau complètement trier pour se rendre contre que la hauteur de l'arbre non équilibré sera la longueur de la liste donc on en déduit que la complexité pire cas de l'accès un élément pour un arbre non équilibré est linéaire en le nombre d'éléments $\mathcal{O}(n)$.

Exercice 3 – Arbres de décisions – Étude de l'algorithme ID3 (Optionnel)

Les arbres de décisions sont un type de méthodes d'apprentissage automatique qui permettent de résoudre des problèmes de classifications. Ces méthodes se positionnent dans le domaine de l'apprentissage supervisé, c'est-à-dire, à partir d'un ensemble de données annotées (valeurs et résultats, p. ex., une image et le contenu représenté) ont construit un modèle qui sera ensuite utilisé pour prédire une propriété sur une donnée n'appartenant pas à l'ensemble d'apprentissage.

Il existe plusieurs méthodes pour construire un arbre de décision, la plus ancienne et la plus simple est l'algorithme ID3 [1] décrit par J.R. Quinlan en 1986. Le principe est relativement simple.

1. On calcule le gain d'information pour chaque attribut
2. Considérant que chaque ligne n'appartiennent pas à la même classe, séparer l'ensemble d'apprentissage S en sous-ensembles en fonction de l'attribut ayant le plus de gain d'information.
3. Créer un nouveau nœud dans l'arbre de décision associé à l'attribut sélectionné avec le gain d'information le grand.
4. Si toutes les lignes du sous-ensemble sont dans la même classe, marquer ce nœud comme feuille avec l'étiquette associée à la classe représentée.

1. https://link.springer.com/article/10.1007/BF00116251

5. Répéter le processus avec les attributs restants jusqu'à que tous les attributs aient été traités ou que l'arbre ait toutes ses feuilles.

Le code proposé dans le répertoire `exo3` donne une mise en œuvre de cet algorithme ID3 (pas fidèlement). Le but de cet exercice est de comprendre comment cet algorithme a été implanté et quelles fonctions/structures de données ont été utilisées pour cela.

Remarque l'objectif n'est pas de comprendre toute la mise en œuvre de l'algorithme ID3 mais de souligner la différence qui peut exister entre une description informelle et une code effectif. Le second objectif est voir une autre utilisation des structures de données arbres dans le domaine de l'apprentissage automatique.

Question 1

Listez les différents types de structures de données qui sont utilisées dans cette mise en œuvre de l'algorithme ID3.

Solution:
Pour cette question, nous ne regardons que le contenu du fichier `decision_tree.c` dans lequel il y a 3 (trois) types de structures de données.
Explication(s):

- ✿ Il y a un arbre binaire dont les valeurs contenues dans les nœuds sont des entiers `integer_bitree_t`
- ✿ Il y a deux files génériques :
  - — une file pour stocker des chaînes de caractères associées à des noms de fichiers. A noter qu'on retrouve les fonctions de constructions et de destruction spécialisées pour les chaînes de caractères ;
  - — une file pour stocker des pointeurs sur des nœuds de l'arbre. A noter que dans ce cas, la fonction d'initialisation de considère pas de fonction de construction et de destruction. Pour ce cas, la raison est que nous laissons au programme principal le soin d'allouer et désallouer la mémoire. On utilise la file uniquement comme un conteneur pour effectuer un parcours en largeur.
- ✿ Il y a un tableau de réels pour stocker des gains d'information durant la construction de l'arbre de décision.

Question 2

L'algorithme ID3 induit beaucoup de manipulation de la base de données d'apprentissage, pour faire des calculs de statistiques ou pour filtrer les lignes de la base de données. Indiquez comme sont mis en œuvre ces manipulations de base de données et listez les différents traitements qui sont effectués.

Solution:
Pour cette question, nous ne regardons le contenu du fichier `decision_tree.c` et le contenu du fichier `utilio.h` dans lesquels on retrouve les différents moyens de manipuler des fichiers au travers de tuyaux (fonction `popen`).
Explication(s):

- ✿ Le fichier `utilio.h` contient en ensemble de fonctions qui permettent d'utiliser les commandes Linux. En particulier, ces fonctions permettent de créer des tuyaux (cf MO101 et l'opérateur `|`).
- ✿ Cette implémentation utilise donc les outils du système pour éviter de mettre en œuvre en langage C des manipulations de fichiers.
- ✿ Les traitements effectués sont essentiellement de la recherche d'information avec les commandes `cut`, `head` et `grep` et des calculs de fréquences avec la commande `wc`. La principale nouveauté par rapport au cours MO101 est l'utilisation de la commande `awk` qui a une capacité intéressante pour manipuler des fichiers au format CSV (ou formatés avec des colonnes), elle est utilisée pour filtrer (sélectionner) les lignes dont une colonne a une certaine valeur.

Question 3

Pourquoi la structure de données arbres binaires est adaptée pour la construction d'arbre de décision pour l'exemple que nous considérons ? Quel impact aurait une autre nature des données sur la structure de données choisies pour représenter l'arbre de décision ?

Solution:
Pour cette question, il faut regarder la structure du fichier `flu/data-flu.csv` pour répondre à la question.
Explication(s):

- ✿ On constate en examinant le fichier `flu/data-flu.csv` que tous les attributs ont des valeurs binaires (présent/absent, Y/N). Cela implique que les nœuds de l'arbre de décisions doivent considérer deux possibilités, la présence/absence d'un attributs d'où un arbre binaire.
- ✿ Si les attributs ont plus de deux valeurs possibles alors l'arbre de décision devient n-aire où n est le nombre de valeurs possibles par attributs.
- ✿ A noter que le nombre de classes dans la classification n'a pas d'impact dans la structure de l'arbre de décision, seulement le type de feuilles sera plus important. Un type de feuille par classe possible.

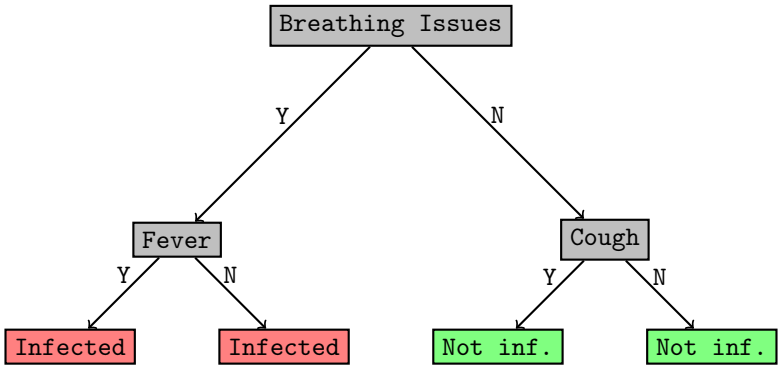
Question 4
Pourquoi la phase d'apprentissage est construite avec une boucle `do-while` ?

Solution:
Pour cette question, il faut comprendre la spécificité de la bibliothèque `libin103` et plus particulièrement les fonctions d'insertion dans les arbres binaires. Voir le fichier `/Library/libin103-1.0/source/integer_bitree.c` pour regarder le code de la fonction `integer_bitree_ins_left`.
Explication(s):

- ✿ Cette structure est imposée par l'API des arbres binaires. En effet, l'insertion dans l'arbre utilise soit la fonction `integer_bitree_ins_left` ou la fonction `integer_bitree_ins_right` qui ont le même comportement quand le pointeur du nœud de l'arbre est `NULL` alors on insère à la racine. Or dans le cas de la construction d'un arbre, on passe forcément par une étape où la racine est `NULL`. La boucle `do-while` permet de passer une phase d'initialisation (durant laquelle la racine est `NULL`) pour ensuite considérer que pointeurs qui ne sont pas `NULL`.

Question 5
Dessinez l'arbre de décision qui est construit à partir de l'ensemble d'apprentissage situé dans le répertoire `flu`.

Solution:



Explication(s):

- ✿ En gris, on a les nœuds associés aux attributs (aux critères de choix)
- ✿ En vert ou rouge, on a les classes (non infecté ou infecté).

POUR S'ENTRAÎNER À LA MAISON

Exercice 1 – Retour sur les ensembles
L'objectif de cet exercice est de programmer une nouvelle version de la structure de données ensembles fondée sur les arbres binaires de recherche équilibrés.

Plus précisément, il faut donner une implémentation des fonctions associées aux ensembles d'entiers :
— initialisation
— destruction

— test d'appartenance
— test d'égalité
— test de sous-ensemble
— union
— intersection
— différence

Fonctions utiles pour cet exercice

- `integer_bistree_init`
- `integer_bistree_destroy`
- `integer_bistree_insert`
- `integer_bistree_remove`
- `integer_bistree_lookup`
- `integer_bistree_size`
- `generic_list_init`
- `generic_inorder`
- `generic_bitree_root`
- `generic_list_head`
- `generic_list_tail`
- `generic_list_next`
- `generic_list_data`

Question 1

Définissez un nouveau type nommé `set_t` qui s'appuie sur les arbres binaires de recherche équilibrés.

Solution:

```
typedef integer_bistree_t set_t;
```

Question 2

Donnez la fonction d'initialisation `init_set` et la fonction de destruction `destroy_set` pour ce nouveau type d'ensemble.

Solution:

```
void init_set (set_t* s) {
  integer_bistree_init (s);
}

void destroy_set (set_t* s) {
  integer_bistree_destroy (s);
}
```

Explication(s):

✿ La structure sous-jacente à ces nouveaux types d'ensembles étant des arbres binaire de recherche équilibrés (AVL), nous définissons simplement les fonctions d'initialisation et de destruction à partir de celles des AVL.

Question 3

Définissez les fonctions d'insertion d'un élément `insert_set` et suppression d'un élément `remove_set` pour ce nouveau type d'ensemble.

Solution:

```
int insert_set (set_t* s, int elem) {
  return integer_bistree_insert (s, elem);
}

int remove_set (set_t* s, int elem) {
  return integer_bistree_remove (s, elem);
}
```

Explication(s):

✿ Comme précédemment, nous utilisons les fonctions d'insertion et de suppression d'un élément qui sont définies pour les AVL, simplifiant grandement la mise en œuvre de ces fonctions pour les ensembles.

Question 4

Définissez la fonction `size_set` qui donne la taille d'un ensemble.

Solution:

```
int size_set (set_t* s) {
  return integer_bistree_size (s);
}
```

Explication(s):

- ✿ On utilise la fonction du même nom associée aux AVL.

Question 5

Définissez la fonction `is_member` qui vérifie si une valeur est présente dans l'ensemble.

Solution:

```
bool is_member (set_t* s, int elem) {
  return integer_bistree_lookup (s, elem);
}
```

Explication(s):

- ✿ La fonction `lookup` des AVL a la même fonction que `is_member`.

Question 6

Définissez une fonction `is_subset` qui indique si un ensemble s_1 est sous-ensemble d'un second ensemble s_2.

Solution:

```
bool is_subset (set_t* s1, set_t* s2) {
  if (size_set (s1) <= size_set (s2)) {

    generic_list_t l;
    generic_list_init (&l, NULL, NULL, NULL);
    generic_bitreenode_t* root = generic_bitree_root (s1);
    generic_inorder (root, &l);

    generic_list_elmt_t *elt;
    for (elt = generic_list_head(&l);
         elt != generic_list_tail(&l);
         elt = generic_list_next (elt)) {

      int v = ((integer_avlnode_t*) generic_list_data(elt))->data;
      if (!is_member (s2, v)) {
        return false;
      }
    }

    return true;
  }
  return false;
}
```

Explication(s):

- ✿ Cette fonction demande un peu plus de travail pour être définie puisqu'il n'y a pas dans l'API des AVL de fonctions qui permet de lister tous les éléments contenus dans l'arbre.
- ✿ En conséquence, nous allons appliquer un algorithme de parcours d'arbre pour extraire une liste de tous les éléments puis parcourir cette liste pour faire un test d'appartenance.
- ✿ Le dernier point de difficulté est que nous devons utiliser une version des listes avec des éléments non typé (`void*`).

Question 7

Définissez une fonction `is_equal` qui vérifie si deux ensembles s_1 et s_2 sont égaux (composé avec les mêmes éléments).

Solution:

```
bool is_equal (set_t* s1, set_t* s2) {
  return is_subset (s1, s2) && is_subset (s2, s1);
}
```

Explication(s):

- ✿ On donne ici une version simple qui utilise la fonction `is_subset` deux fois.

Question 8

Définissez la fonction `union_set` qui calcule l'union de deux ensembles s_1 et s_2. Le résultat est rendu par effet de bord (par un pointeur).

Solution:

```
int union_set (set_t* su, set_t* s1, set_t* s2) {
  init_set (su);

  generic_list_t l;
  generic_list_init (&l, NULL, NULL, NULL);
  generic_bitreenode_t* root = generic_bitree_root (s1);
  generic_inorder (root, &l);

  generic_list_elmt_t *elt;
  for (elt = generic_list_head(&l);
       elt != NULL;
       elt = generic_list_next (elt)) {

    int v = ((integer_avlnode_t*) generic_list_data(elt))->data;
    insert_set (su, v);
  }

  root = generic_bitree_root (s2);
  generic_list_destroy (&l);
  generic_list_init (&l, NULL, NULL, NULL);
  generic_inorder (root, &l);

  for (elt = generic_list_head(&l);
       elt != NULL;
       elt = generic_list_next (elt)) {

    int v = ((integer_avlnode_t*) generic_list_data(elt))->data;
    insert_set (su, v);
  }

  return 0;
}
```

Explication(s):

- ✿ Comme pour la fonction `is_subset` nous devons énumérer tous les éléments des deux ensembles pour composer le résultat.
- ✿ Nous faisons donc deux parcours d'arbre pour obtenir les listes des éléments des deux ensembles et les ajouter dans le troisième ensemble.

Question 9

Définissez la fonction `difference_set` qui calcule la différence de deux ensembles s_1 et s_2. Le résultat est rendu par effet de bord (par un pointeur).

Solution:

```
int difference_set (set_t* sd, set_t* s1, set_t* s2) {
  init_set (sd);

  generic_list_t l;
  generic_list_init (&l, NULL, NULL, NULL);
  generic_bitreenode_t* root = generic_bitree_root (s1);
  generic_inorder (root, &l);

  generic_list_elmt_t *elt;
  for (elt = generic_list_head(&l);
       elt != NULL;
       elt = generic_list_next (elt)) {

    int v = ((integer_avlnode_t*) generic_list_data(elt))->data;
    if (!is_member (s2, v)) {
      insert_set (sd, v);
    }
  }

  return 0;
}
```

Explication(s):

- ✿ Comme pour la fonction `is_subset` nous devons énumérer tous les éléments d'un ensemble.
- ✿ Nous faisons donc un parcours d'arbre pour obtenir la liste des éléments de s_1 et on ajoute dans le résultat les éléments de s_1 qui n'appartiennent pas à s_2.

Question 10

Définissez la fonction `intersection_set` qui calcule l'intersection de deux ensembles s_1 et s_2. Le résultat est rendu par effet de bord (par un pointeur).

Solution:

```
int intersection_set (set_t* si, set_t* s1, set_t* s2) {
  init_set (si);

  generic_list_t l;
  generic_list_init (&l, NULL, NULL, NULL);
  generic_bitreenode_t* root = generic_bitree_root (s1);
  generic_inorder (root, &l);

  generic_list_elmt_t *elt;
  for (elt = generic_list_head(&l);
       elt != NULL;
       elt = generic_list_next (elt)) {

    int v = ((integer_avlnode_t*) generic_list_data(elt))->data;
    if (is_member (s2, v)) {
      insert_set (si, v);
    }
  }

  return 0;
}
```

Explication(s):

- ✿ Comme pour la fonction `is_subset` nous devons énumérer tous les éléments d'un ensemble.
- ✿ Nous faisons donc un parcours d'arbre pour obtenir la liste des éléments de s_1 et on ajoute dans le résultat les éléments de s_1 qui appartiennent à s_2.

Question 11

Discutez des complexités des fonctions `insert_set`, `remove_set`, `union_set`, `difference_set` et `intersection_set`.

Solution:
TBD