



Apprendre à programmer les arbres en langage C - Première partie

Implémentation d'un arbre simple

Table des matières

- I. Introduction
- II. Un arbre binaire de recherche
- III. Codes sources de l'exemple
- IV. Conclusion et remerciements

Les arbres servent à mémoriser des données. Ils sont constitués d'éléments que l'on appelle souvent des nœuds (node). Ils sont semblables aux listes chaînées par le fait que les éléments sont chaînés les uns avec les autres, mais avec la possibilité que plusieurs branches partent d'un nœud, d'où leur nom (on pourrait très bien voir une liste chaînée comme un arbre à une seule branche). Il est courant d'appeler le premier élément d'un arbre la racine. La racine est un nœud qui n'a pas de parent. On peut aussi entendre parler de feuilles, ce sont les nœuds qui sont au bout des branches et qui n'ont donc pas d'enfants.

Ce tutoriel va aborder les arbres binaires.

Pour réagir au contenu de ce tutoriel, un espace de dialogue vous est proposé sur le forum :
7 commentaires 🌟🌟🌟🌟🌟

Article lu 106365 fois.

L'auteur

CGI 🧑

L'article

Publié le 14 février 2016

Version PDF Version hors-ligne

ePub, Azw et Mobi

Liens sociaux



I. Introduction ▲

Je ne vais pas vous faire un cours complet sur les arbres, mais vous montrer comment les implémenter en langage C.

Voyons tout de même quelques notions élémentaires pour qu'il n'y ait pas d'ambiguïtés :

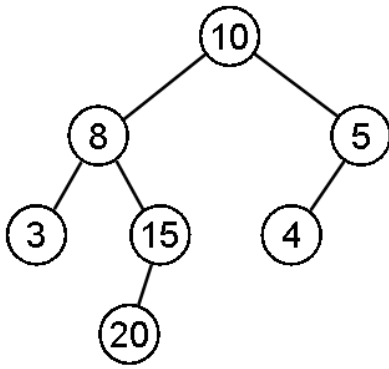
Qu'est-ce qu'un arbre ?

Tout comme les listes chaînées, les arbres servent à mémoriser des données. Ils sont constitués d'éléments que l'on appelle souvent des nœuds (node). Ils sont semblables aux listes chaînées par le fait que les éléments sont chaînés les uns avec les autres, mais avec la possibilité que plusieurs branches partent d'un nœud, d'où leur nom (on pourrait très bien voir une liste chaînée comme un arbre à une seule branche). Il est courant d'appeler le premier élément d'un arbre la racine. La racine est un nœud qui n'a pas de parent. On peut aussi entendre parler de feuilles, ce sont les nœuds qui sont au bout des branches et qui n'ont donc pas d'enfants.

Ce tutoriel étant destiné à aborder les arbres, nous allons donc en créer un qui sera le plus simple possible, ce sera un arbre binaire.

Un arbre binaire est un arbre où chaque nœud peut avoir au maximum deux branches. Pour différencier les branches, on les nomme souvent droite ou gauche (right, left).

On peut se faire un petit schéma pour se le représenter visuellement.



La racine en haut et les branches vers le bas, désolé, mais c'est la représentation la plus courante pour les arbres (informatique).

Pour qu'un arbre soit efficace, il ne faut pas le remplir anarchiquement, mais de façon ordonnée, ceci afin de retrouver nos données rapidement et sans avoir à parcourir l'arbre complet. C'est là son gros avantage par rapport aux listes chaînées. Il est souvent bien plus rapide de parcourir l'arbre de la racine jusqu'à une feuille, plutôt qu'une longue liste chaînée parfois entièrement.

II. Un arbre binaire de recherche ▲

C'est un des arbres les plus simples et nous allons le simplifier au maximum. Ses éléments (ou nœuds) ne contiendront qu'une valeur de type entier (on aurait pu y embarquer des structures de données plus complexes, mais ça ne nous est pas utile et ça surchargerait l'exemple). C'est cette valeur qui nous servira à ordonner les éléments. Nous l'appellerons donc la clé (key).

Tout comme les listes chaînées, les arbres sont basés sur une structure du langage C. La différence sera qu'elle contiendra deux pointeurs pour lier les éléments, un pointeur pour accéder à la branche de gauche et l'autre pour accéder à la branche de droite. Nous avons maintenant suffisamment d'éléments pour constituer la structure d'un nœud.

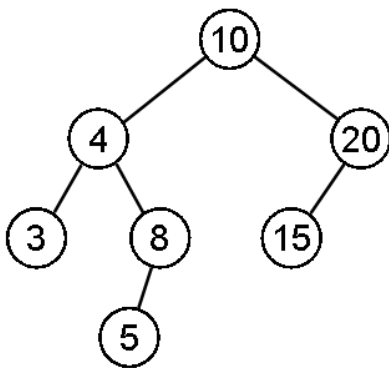
Sélectionnez

- 1.
- 2.
- 3.
- 4.
- 5.
- 6.

```
typedef struct node
{
    unsigned int key;
    struct node *left;
    struct node *right;
} node ;
```

La deuxième étape est de trier les éléments à leur insertion dans l'arbre. Le tri sera effectué sur la valeur de la clé (key). Le premier élément est inséré à la racine de l'arbre, l'élément suivant est inséré à gauche si la valeur de sa clé est inférieure à celle de la racine et à droite si la valeur de sa clé est supérieure à celle de la racine (on aurait pu faire l'inverse). Pour les éléments qui suivent, c'est le même principe jusqu'à trouver un emplacement libre au bout d'une branche.

Par exemple, si on avait inséré des éléments ayant comme clé : 10, 20, 4, 8, 5, 15, 3 dans cet ordre, on aurait un arbre équivalent au schéma suivant :



Comme vous le voyez, il va être facile de retrouver un élément, il suffira de suivre le même cheminement que pour l'insertion.

Pour notre exemple, le point d'entrée de l'arbre sera un pointeur initialisé à NULL, qui devra pointer sur la racine dès l'insertion du premier élément.

Sélectionnez

- 1.

```
node *Arbre = NULL;
```

La première chose que l'on peut faire, c'est donc d'insérer des éléments. Pour cela, nous allons créer la fonction addNode.

Sélectionnez

- 1.
- 2.
- 3.
- 4.
- 5.
- 6.

7.
8.
9.
10.
11.
12.
13.
14.
15.
16.
17.
18.
19.
20.
21.
22.
23.
24.
25.
26.
27.
28.

```
void addNode(node **tree, unsigned int key)
{
    node *tmpNode;
    node *tmpTree = *tree;

    node *elem = malloc(sizeof(node));
    elem->key = key;
    elem->left = NULL;
    elem->right = NULL;

    if(tmpTree)
    do
    {
        tmpNode = tmpTree;
        if(key > tmpTree->key )
        {
            tmpTree = tmpTree->right;
            if(!tmpTree) tmpNode->right = elem;
        }
        else
        {
            tmpTree = tmpTree->left;
            if(!tmpTree) tmpNode->left = elem;
        }
    }
    while(tmpTree);
    else *tree = elem;
}
```

Son rôle est de créer l'élément à l'aide de la fonction malloc, d'initialiser ses champs : la clé avec la valeur désirée (passé en paramètre à la fonction), d'initialiser les deux pointeurs à NULL (ils seront positionnés au bout d'une branche et n'ont donc pas d'enfants) et de les insérer dans l'arbre en tenant compte des critères de tri. Donc si l'élément n'est pas le premier, on boucle (boucle do while) afin d'avancer de nœud en nœud jusqu'à atteindre un emplacement libre (pointeur à NULL) et à chaque nœud on part à droite, si la clé est supérieure à celle du nœud courant, ou à gauche si elle est inférieure ou égale à celle du nœud courant.

Le cas du premier élément (racine de l'arbre) impose d'affecter l'adresse de cet élément au pointeur sur l'arbre que l'on a passé en paramètre à la fonction. Il s'impose donc que ce paramètre soit un pointeur de pointeur, afin de passer l'adresse du pointeur sur l'arbre à la fonction. Fonction que l'on appellera donc de la façon suivante :

Sélectionnez

1.
2.
3.

```
node *Arbre = NULL;
//...
addNode(&Arbre, 30);
```

Comme on peut le remarquer, ce n'est guère plus compliqué que pour une liste chaînée.

Remarque : je n'ai pas testé les retours de la fonction malloc pour ne pas surcharger le code.

Nous avons dit que notre arbre est un arbre de recherche. C'est donc la deuxième fonction que nous allons créer. Elle devra nous indiquer si un élément avec une clé de valeur x est présent dans l'arbre.

Sélectionnez

1.
2.
3.
4.
5.
6.
7.
8.
9.
10.
11.

```
int searchNode(node *tree, unsigned int key)
{
    while(tree)
    {
        if(key == tree->key) return 1;

        if(key > tree->key ) tree = tree->right;
        else tree = tree->left;
    }
    return 0;
}
```

Le principe est identique à la fonction d'insertion. On suit le cheminement en partant à droite ou à gauche selon la valeur de la clé. À chaque nœud on vérifie si on est en présence de l'élément recherché, si oui on retourne la valeur 1. Quand on arrive au bout de la branche si on ne l'a pas trouvé on retourne 0. On est certain qu'il ne se trouve pas dans une autre branche, il n'y a donc pas besoin de tester.

Si la structure était plus complexe, nous aurions pu faire retourner un contenu par la fonction. Par exemple une valeur si on avait eu un couple clé/valeur.

L'appel de cette fonction est des plus simples :

Sélectionnez

1.

```
if(searchNode(Arbre, Key)) //...
```

À ce stade nous avons déjà un arbre opérationnel.

Mais nous n'allons pas nous arrêter là, nous allons lui ajouter une fonction qui permettra d'afficher l'arbre et de trier, en plus.

Pour cela il va falloir parcourir l'arbre complet, et là nous n'avons guère le choix si l'on veut faire cela de façon simple, nous devons utiliser des fonctions récursives (une fonction récursive est une fonction qui s'appelle elle-même).

Sélectionnez

1.

2.

3.

4.

5.

6.

7.

8.

9.

10.

```
void printTree(node *tree)
{
    if(!tree) return;

    if(tree->left) printTree(tree->left);

    printf("Cle = %d\n", tree->key);

    if(tree->right) printTree(tree->right);
}
```

Dans la fonction, on remarque deux appels récursifs de la fonction, un sur le pointeur de gauche et l'autre sur le pointeur de droite de l'élément en cours de traitement. Ce qui va permettre de passer d'élément en élément.

Ce qui fera arrêter la récursivité dans notre fonction, c'est le test du pointeur NULL des feuilles.

Pour afficher les valeurs des clés dans l'ordre, il faut partir à **gauche toute** avant d'afficher la clé au retour de la fonction, par contre si l'on part à droite, on affiche la clé en cours avant l'appel de la fonction (vous pouvez essayer de suivre le cheminement sur le schéma).

Allez ! Une autre fonction d'affichage très semblable à la précédente, sauf que l'on part à **droite toute** en premier.

Sélectionnez

1.

2.

3.

4.

5.

6.

7.

8.

9.

10.

```
void printReverseTree(node *tree)
{
    if(!tree) return;

    if(tree->right) printReverseTree(tree->right);

    printf("Cle = %d\n", tree->key);

    if(tree->left) printReverseTree(tree->left);
}
```

Vous avez testé ! Cela affiche l'arbre trié en sens inverse tout simplement.

Une dernière fonction nécessaire dans notre exemple : nous avons alloué de la mémoire avec malloc, il faut donc la libérer. Là aussi, il faut parcourir l'arbre complet, nous utiliserons donc le même principe que pour les fonctions d'affichage.

Sélectionnez

1.
2.
3.
4.
5.
6.
7.
8.
9.
10.
11.
12.
13.
14.

```
void clearTree(node **tree)
{
    node *tmpTree = *tree;

    if(!tree) return;

    if(tmpTree->left) clearTree(&tmpTree->left);

    if(tmpTree->right) clearTree(&tmpTree->right);

    free(tmpTree);

    *tree = NULL;
}
```

L'utilisation est des plus simples. Je mets un code d'utilisation en bas de page (main.c) qui vaut mieux qu'un grand discours.

III. Codes sources de l'exemple▲

Pour plus de lisibilité et de possibilités de réutilisation de cet arbre, nous séparerons son code de son utilisation.

rbtree.h :

Sélectionnez

1.
2.
3.
4.
5.
6.
7.
8.
9.
10.
11.
12.
13.
14.
15.
16.
17.
18.
19.
20.
21.
22.
23.
24.
25.
26.
27.
28.
29.

```

#ifndef CGI_RBTREE_H
#define CGI_RBTREE_H

typedef struct node
{
    unsigned int key;
    struct node *left;
    struct node *right;
} node ;

#ifdef __cplusplus
extern "C" {
#endif

    void addNode(node **tree, unsigned int key);

    int searchNode(node *tree, unsigned int key);

    void printTree(node *tree);

    void printReverseTree(node *tree);

    void clearTree(node **tree);

#ifdef __cplusplus
}
#endif

#endif //CGI_RBTREE_H
rbtree.c :

```

Sélectionnez

- 1.
- 2.
- 3.
- 4.
- 5.
- 6.
- 7.
- 8.
- 9.
- 10.
- 11.
- 12.
- 13.
- 14.
- 15.
- 16.
- 17.
- 18.
- 19.
- 20.
- 21.
- 22.
- 23.
- 24.
- 25.
- 26.
- 27.
- 28.
- 29.
- 30.
- 31.
- 32.
- 33.
- 34.
- 35.
- 36.
- 37.
- 38.
- 39.
- 40.
- 41.
- 42.
- 43.
- 44.
- 45.
- 46.
- 47.
- 48.
- 49.
- 50.
- 51.
- 52.
- 53.
- 54.
- 55.

56.
57.
58.
59.
60.
61.
62.
63.
64.
65.
66.
67.
68.
69.
70.
71.
72.
73.
74.
75.
76.
77.
78.
79.
80.
81.
82.
83.
84.
85.
86.
87.
88.
89.

```

#include <stdio.h>
#include <stdlib.h>
#include "rbtree.h"

void addNode(node **tree, unsigned int key)
{
    node *tmpNode;
    node *tmpTree = *tree;

    node *elem = malloc(sizeof(node));
    elem->key = key;
    elem->left = NULL;
    elem->right = NULL;

    if(tmpTree)
    do
    {
        tmpNode = tmpTree;
        if(key > tmpTree->key )
        {
            tmpTree = tmpTree->right;
            if(!tmpTree) tmpNode->right = elem;
        }
        else
        {
            tmpTree = tmpTree->left;
            if(!tmpTree) tmpNode->left = elem;
        }
    }
    while(tmpTree);
    else *tree = elem;
}

/*****/

int searchNode(node *tree, unsigned int key)
{
    while(tree)
    {
        if(key == tree->key) return 1;

        if(key > tree->key ) tree = tree->right;
        else tree = tree->left;
    }
    return 0;
}

/*****/

void printTree(node *tree)
{
    if(!tree) return;

    if(tree->left) printTree(tree->left);

    printf("Cle = %d\n", tree->key);

    if(tree->right) printTree(tree->right);
}

/*****/

void printReverseTree(node *tree)
{
    if(!tree) return;

    if(tree->right) printReverseTree(tree->right);

    printf("Cle = %d\n", tree->key);

    if(tree->left) printReverseTree(tree->left);
}

/*****/

void clearTree(node **tree)
{
    node *tmpTree = *tree;

    if(!tree) return;

    if(tmpTree->left) clearTree(&tmpTree->left);

    if(tmpTree->right) clearTree(&tmpTree->right);

    free(tmpTree);

    *tree = NULL;
}

```



```
} _____
```

Voici un exemple d'utilisation de l'arbre que nous venons de construire.

main.c :

Sélectionnez

- 1.
- 2.
- 3.
- 4.
- 5.
- 6.
- 7.
- 8.
- 9.
- 10.
- 11.
- 12.
- 13.
- 14.
- 15.
- 16.
- 17.
- 18.
- 19.
- 20.
- 21.
- 22.
- 23.
- 24.
- 25.
- 26.
- 27.
- 28.
- 29.
- 30.
- 31.
- 32.
- 33.
- 34.
- 35.
- 36.
- 37.
- 38.
- 39.
- 40.
- 41.
- 42.
- 43.
- 44.
- 45.
- 46.

```

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

#include "rbtree.h"

int main()
{
    unsigned int Key;
    node *Arbre = NULL;

    addNode(&Arbre, 30);
    addNode(&Arbre, 20);
    addNode(&Arbre, 50);
    addNode(&Arbre, 45);
    addNode(&Arbre, 25);
    addNode(&Arbre, 80);
    addNode(&Arbre, 40);
    addNode(&Arbre, 70);
    addNode(&Arbre, 25);
    addNode(&Arbre, 10);
    addNode(&Arbre, 60);

    puts("-----");

    printTree(Arbre);

    puts("-----");

    printReverseTree(Arbre);

    puts("-----");

    Key = 30;
    if(searchNode(Arbre, Key)) printf("La cle %d existe.\n", Key);
    else printf("La cle %d n'existe pas.\n", Key);

    Key = 32;
    if(searchNode(Arbre, Key)) printf("La cle %d existe.\n", Key);
    else printf("La cle %d n'existe pas.\n", Key);

    puts("-----");

    clearTree(&Arbre);

    return 0;
}

```

Cet arbre est bien adapté pour la recherche d'élément, l'insertion et la recherche ne nécessitant pas de parcourir l'arbre en entier. Il peut aussi faire du tri, vous l'avez vu avec les fonctions d'affichage. Il est même assez performant si les valeurs des clés des éléments insérés sont aléatoires.

Par contre, il devient très mauvais dans le cas d'insertion d'éléments déjà triés ou partiellement triés. Dans ce cas au lieu de faire plusieurs branches il pourrait en faire une seule, très grande dans le pire des cas (ce qui reviendrait à une liste chaînée).


Ce qui nous amènera à parler d'un autre type d'arbre au chapitre suivant, les **arbres tassés**.

Bonne lecture,

CGI.

IV. Conclusion et remerciements ▲

Nous tenons à remercier Jacques THERY pour la relecture orthographique et Malick SECK pour la mise au gabarit.

Vous avez aimé ce tutoriel ? Alors partagez-le en cliquant sur les boutons suivants : 

Copyright © 2016 CGI. Aucune reproduction, même partielle, ne peut être faite de ce site ni de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

<p>Augmentation de la popularité de C#, selon l'indice Tiobe</p>	<p>Linus Torvalds se prépare à faire passer le noyau Linux au C moderne (C11)</p>	<p>JetBrains lance le programme d'accès anticipé (EAP) à CLion 2022.2, la deuxième mise à jour majeure de l'année de son EDI C/C++ multiplateforme</p>	<p>Apprendre à installer Gtk+ et Code::Blocks sous Windows, un tutoriel de Gérard Dumas</p>
---	--	---	--

Contactez le responsable de la rubrique C

