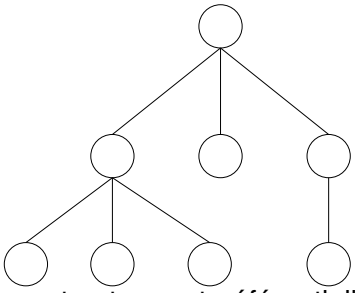
# Programmation et structures de données en C cours 4: Arbres

Jean-Lou Desbarbieux, Stéphane Doncieux
et Mathilde Carpentier
LU2IN018 Sorbonne Université 2022/2023

## Sommaire



## Définition d'un arbre



L'arbre est une structure autoréférentielle, c'est une généralisation de la liste : alors qu'un élément de liste a un seul successeur (l'élément suivant) dans un arbre il peut y avoir plusieurs sucesseurs.

## Notion de noeud

L'élément de base d'un arbre s'appelle le noeud. C'est une structure composée de :

- un champ de donnée ;
- plusieurs pointeurs vers des noeuds.

## Terminologie

On utilise classiquement la terminologie suivante pour décrire les arbres :

**Père** Le père A d'un noeud B est l'unique noeud tel que :
- un des sous-arbres contient B ;
- hauteur(A) - hauteur(B) = 1.

**fils** Les fils d'un noeud A sont les noeuds qui ont A pour père.

**frères** Les frères d'un noeud A sont les noeuds qui possèdent le même père que A.

**Sous-arbre** Un sous-arbre d'un noeud A est un arbre dont la racine est un fils de A.

## Terminologie suite

**Racine** La racine de l'arbre est l'unique noeud qui n'a pas de père.

**Feuille** Une feuille est un noeud qui n'a pas de fils.

**Hauteur** Nombre maximum d'arêtes entre la racine et une feuille.

## Arbres n-aires

Un arbre dont les noeuds ont au plus n fils est un arbre n-aire. Lorsque n vaut 2, l'arbre est dit binaire. Dans le cas particulier de l'arbre binaire on utilise les termes de fils gauche et fils droit d'un noeud, ainsi que les notions de sous-arbre gauche et de sous-arbre droit.

## Déclaration du noeud

```
typedef struct _un_noeud *P_un_noeud;
typedef struct _un_noeud {
  int data;
  P_un_noeud gauche;
  P_un_noeud droit;
} Un_noeud;
```

## Arbre binaire ”simple” : fonctions

- création d'un noeud
- ajouter un noeud
- détruire l'arbre
- afficher l'arbre

## Arbre binaire ”simple” : créer un noeud

```
P_un_noeud creer_noeud(int data) {
  P_un_noeud p=(P_un_noeud)malloc(sizeof(Un_noeud));
  if (p==NULL) {
    fprintf(stderr,"Erreur lors de l'allocation d'un noe
    return NULL;
  }
  p->data=data;
  p->gauche=NULL;
  p->droit=NULL;
  return p;
}
```

## Arbre binaire ”simple” : ajouter à la racine

```
P_un_noeud ajouter_racine(int data, P_un_noeud abg, P_un_noeu
  P_un_noeud r=creer_noeud(data);
  r->gauche=abg;
  r->droit=abd;
  return r;
}
```

## Arbre binaire ”simple” : détruire l'arbre

```
void detruire_noeud(P_un_noeud p) {
  free(p);
}

void detruire_arbre(P_un_noeud a) {
  if (a!=NULL) {
    detruire_arbre(a->gauche);
    detruire_arbre(a->droit);
    detruire_noeud(a);
  }
}
```

## Arbre binaire ”simple” : afficher l’arbre

```
void afficher_arbre(P_un_noeud a) {
  if (a) {
    printf("( %d ",a->data);
    afficher_arbre(a->gauche);
    afficher_arbre(a->droit);
    printf(")");
  }
}
```

## Arbre binaire ”simple” : utilisation

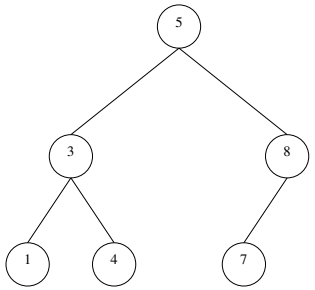
```
#include <stdio.h>
#include "arbre.h"

int main(void) {
  P_un_noeud p1 = ajouter_racine(3, NULL, NULL);
  P_un_noeud p2 = ajouter_racine(4, NULL, NULL);
  P_un_noeud p3 = ajouter_racine(5, p1, p2);
  afficher_arbre(p3);
  printf("\n");
  detruire_arbre(p3);
  return 0;
}
```

Résultat attendu :

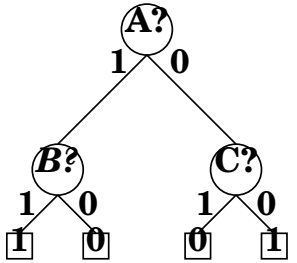
```
bash$ ./arbre
( 5 ( 3 )( 4 ))
```

## Arbres binaires de recherche (ABR)




Tous les noeuds du sous-arbre de gauche ont une valeur inférieure.
Tous les noeuds du sous-arbre de droite ont une valeur supérieure ou égale.

## Arbres de décision binaire (BDD)



## Parcours en profondeur

Le parcours dit en profondeur d'abord consiste à explorer un arbre de haut en bas puis de gauche à droite.
On distingue trois types de parcours en profondeurs :

- le parcours préfixe traite le noeud courant puis le sous-arbre gauche puis le sous-arbre droit.
- le parcours infixe traite le sous-arbre gauche puis le noeud courant puis le sous-arbre droit.
- le parcours postfixe traite le sous-arbre gauche puis le sous-arbre droit puis le noeud courant.

## Parcours en profondeur recursif prefixe

Affichage du contenu d'un arbre binaire de recherche.

```
void aff_prof_prefixe(P_un_noeud a) {

  if (a) {
    printf("␣%d",a->data);
    aff_prof_prefixe(a->gauche);
    aff_prof_prefixe(a->droit);
  }
}
```

## Parcours en profondeur recursif infixe

```
void aff_prof_infixe(P_un_noeud a) {

  if (a) {
    aff_prof_infixe(a->gauche);
    printf("␣%d",a->data);
    aff_prof_infixe(a->droit);
  }
}
```

## Parcours en profondeur recursif postfixe

```
void aff_prof_postfixe(P_un_noeud a) {

  if (a) {
    aff_prof_postfixe(a->gauche);
    aff_prof_postfixe(a->droit);
    printf("␣%d",a->data);
  }
}
```

## Parcours en profondeur itératif prefixe

```
void aff_prof_prefixe_iteratif(P_un_noeud a) {
  P_un_noeud p = a;
  T_pile pile = NULL;
  if (p) {
    pile=push_p(pile, p);
    do
      {
        pile=pop_p(pile, &p);
        printf("%d ", p->data);
        if(p->droit) {
          pile=push_p(pile, p->droit);
        }
        if(p->gauche){
          pile=push_p(pile, p->gauche);
        }
      }
    while(pile);
  }
}
```

## Parcours en largeur

```
void aff_largeur_iteratif(P_un_noeud a)
{
  P_un_noeud p = a;
  T_file fifo = NULL;
  if (p) {
    fifo=push_f( fifo , p);
    do
      {
        fifo=pop_f(fifo , &p);
        printf("%d ", p->data);
        if(p->gauche) {
          fifo=push_f(fifo , p->gauche);
        }
        if(p->droit) {
          fifo=push_f(fifo, p->droit);
        }
      }
    while(fifo);
  }
}
```

# C'est tout pour aujourd'hui !