Programmation Avancée

Cours 6 : Les arbres

Simon Forest

4 mars 2021

Sommaire

Définitions et exemples

Arbres binaires

Arbres généraux

Nœuds de types différents

Types de parcours

Sommaire

Définitions et exemples

Arbres binaires

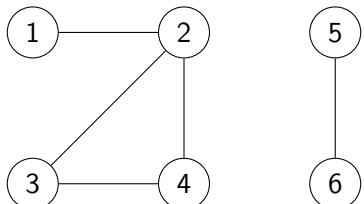
Arbres généraux

Nœuds de types différents

Types de parcours

Graphes

Un **graphe** est un ensemble de nœuds et d'arêtes entre ces nœuds.



Ici un graphe avec 6 nœuds et 5 arêtes.

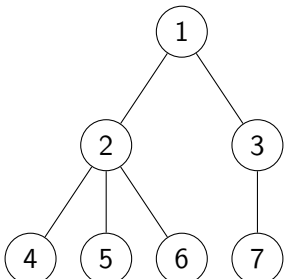
On voit que ce graphe admet un **cycle** : 2 - 3 - 4.

De plus, ce graphe **n'est pas connexe** : on ne peut pas rejoindre le nœud 5 à partir du nœud 2 en suivant des arêtes.

Arbres

Un arbre est un graphe qui est sans cycles et connexe avec de plus un nœud distingué qui est la **racine**.
Les arbres se représentent naturellement en mettant la racine tout en haut et en organisant les autres nœuds suivant leurs distances à la racine.

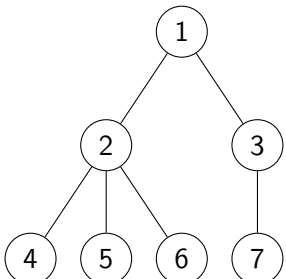
Exemple et terminologie



Ceci est un graphe sans cycles et connexe.

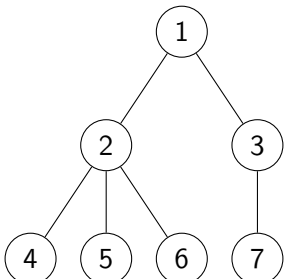
En prenant le nœud 1 comme racine, on obtient donc un arbre.

Exemple et terminologie



Chaque nœud d'un arbre a alors une **profondeur**, qui est sa distance à la racine (combien d'arêtes faut-il traverser depuis la racine pour atteindre ce nœud).

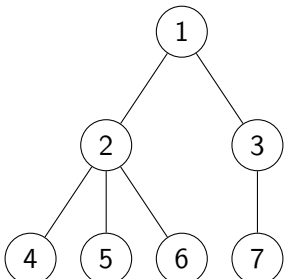
Exemple et terminologie



Chaque nœud d'un arbre a alors une **profondeur**, qui est sa distance à la racine (combien d'arêtes faut-il traverser depuis la racine pour atteindre ce nœud).

Le nœud 1 est de profondeur 0.

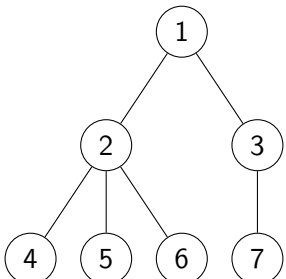
Exemple et terminologie



Chaque nœud d'un arbre a alors une **profondeur**, qui est sa distance à la racine (combien d'arêtes faut-il traverser depuis la racine pour atteindre ce nœud).

Les nœuds 2 et 3 sont de profondeur 1.

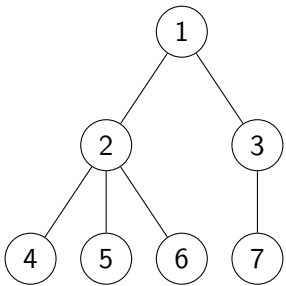
Exemple et terminologie



Chaque nœud d'un arbre a alors une **profondeur**, qui est sa distance à la racine (combien d'arêtes faut-il traverser depuis la racine pour atteindre ce nœud).

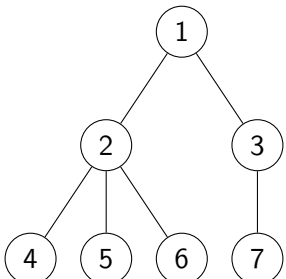
Les nœuds 4,5,6 et 7 sont de profondeur 2.

Exemple et terminologie



La **hauteur** d'un arbre est la profondeur de son nœud le plus profond.

Exemple et terminologie

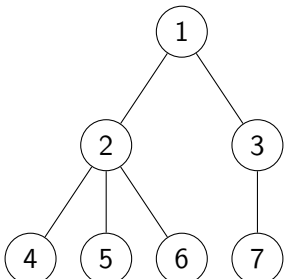


La **hauteur** d'un arbre est la profondeur de son nœud le plus profond.

Les nœuds les plus profonds sont ici les nœuds 4, 5, 6 et 7 de profondeur 2.

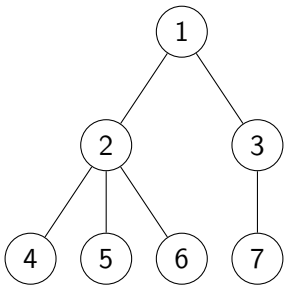
La hauteur de l'arbre est donc 2.

Exemple et terminologie



On remarque que chaque nœud, sauf le nœud racine, a un nœud juste au dessus de lui, appelé **nœud parent**.

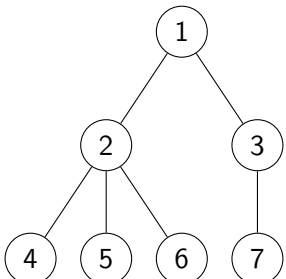
Exemple et terminologie



On remarque que chaque nœud, sauf le nœud racine, a un nœud juste au dessus de lui, appelé **nœud parent**.

Le nœud parent des nœuds 4,5 et 6 est 2.

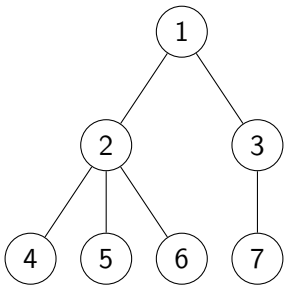
Exemple et terminologie



On remarque que chaque nœud, sauf le nœud racine, a un nœud juste au dessus de lui, appelé **nœud parent**.

Le nœud parent du nœud 7 est 3.

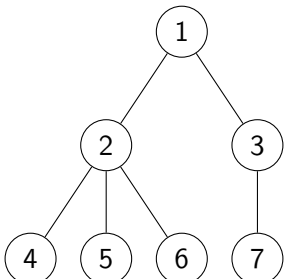
Exemple et terminologie



On remarque que chaque nœud, sauf le nœud racine, a un nœud juste au dessus de lui, appelé **nœud parent**.

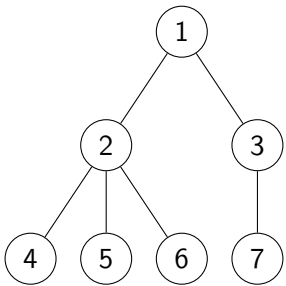
Les nœuds parents des nœuds 2 et 3 est 1.

Exemple et terminologie



Symétriquement, chaque nœud est parent d'un certain nombre de nœuds, qui sont ses **enfants**. On appelle ce nombre le **degré** du nœud.

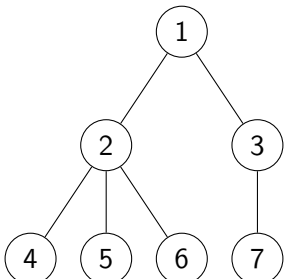
Exemple et terminologie



Symétriquement, chaque nœud est parent d'un certain nombre de nœuds, qui sont ses **enfants**. On appelle ce nombre le **degré** du nœud.

Les nœuds 4,5,6 et 7 n'ont pas d'enfants. Leur degré est donc 0.

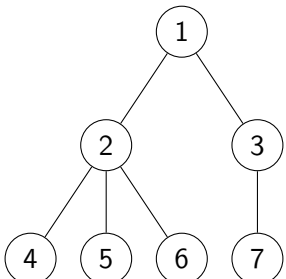
Exemple et terminologie



Symétriquement, chaque nœud est parent d'un certain nombre de nœuds, qui sont ses **enfants**. On appelle ce nombre le **degré** du nœud.

Le nœud 2 a trois enfants : les nœuds 4,5 et 6. Son degré est donc 3.

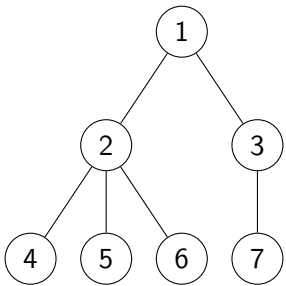
Exemple et terminologie



Symétriquement, chaque nœud est parent d'un certain nombre de nœuds, qui sont ses **enfants**. On appelle ce nombre le **degré** du nœud.

Le nœud 3 a un enfant : le nœud 7. Son degré est donc 1.

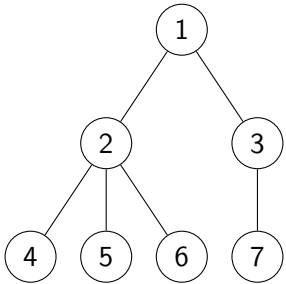
Exemple et terminologie



Symétriquement, chaque nœud est parent d'un certain nombre de nœuds, qui sont ses **enfants**. On appelle ce nombre le **degré** du nœud.

Le nœud 1 a deux enfant : les nœuds 2 et 3. Son degré est donc 2.

Exemple et terminologie



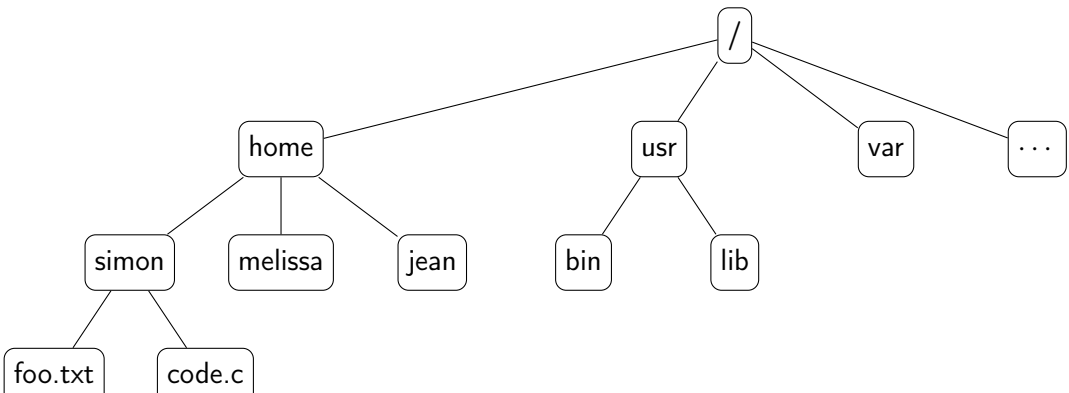
Un nœud qui ne possède pas d'enfants est une **feuille**.

Ici, les nœuds 4, 5, 6 et 7 sont des feuilles.

Exemples concrets

Les arbres sont une structure assez fréquente en informatique.

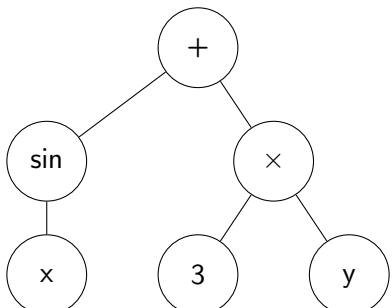
Par exemple, pour représenter l'arborescence d'un système de fichiers.



Exemples concrets

Les arbres sont une structure assez fréquente en informatique.

Par exemple, pour représenter des expressions arithmétiques.

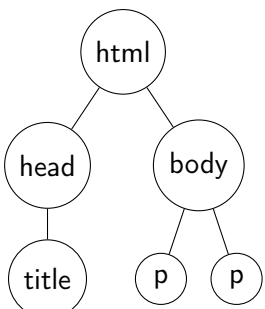


Arbre de l'expression $\sin(x) + 3 \times y$.

Exemples concrets

Les arbres sont une structure assez fréquente en informatique.

Par exemple, pour représenter la structure d'une page HTML.




```
<html>
  <head>
    <title>
    Un titre
    </title>
  </head>
  <body>
    <p>Premier paragraphe.</p>
    <p>Deuxième paragraphe.</p>
  </body>
</html>
```

Sommaire

Définitions et exemples

Arbres binaires

Arbres généraux

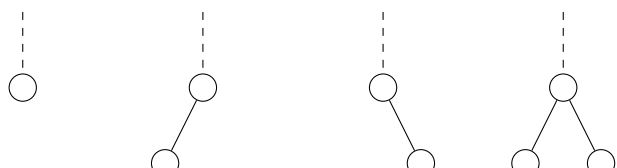
Nœuds de types différents

Types de parcours

Arbres binaires

Avant de voir comment représenter les arbres généraux, on va traiter un cas particulier : celui des **arbres binaires**.

Un arbre binaire est un arbre où chaque peut avoir jusqu'à deux nœuds enfants : un **nœud gauche** et un **nœud droit**.

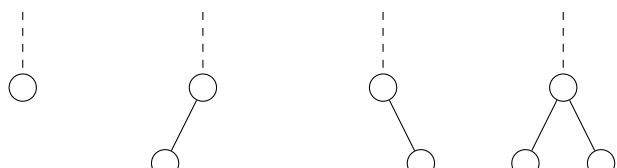


Le **sous-arbre gauche** d'un nœud est le sous-arbre dont la racine est le nœud gauche de ce nœud.

Arbres binaires

Avant de voir comment représenter les arbres généraux, on va traiter un cas particulier : celui des **arbres binaires**.

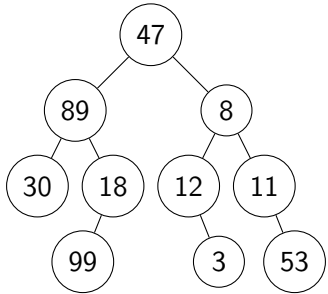
Un arbre binaire est un arbre où chaque peut avoir jusqu'à deux nœuds enfants : un **nœud gauche** et un **nœud droit**.

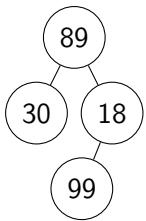


Le **sous-arbre droit** d'un nœud est le sous-arbre dont la racine est le nœud droit de ce nœud.

Exemple

Un exemple d'arbre binaire qui contient des entiers :



On a que  est le sous-arbre gauche de la racine.

Arbres binaire en C

Pour représenter un arbre binaire en C, on utilise un type faisant intervenir de la **récursivité** :

```c
typedef struct arb_bin {
  int val;
  struct arb_bin *gauche, *droite;
  // ^ pointeurs sur le type en train d'être défini
} arb_bin;
```

Arbres binaire en C

Pour représenter un arbre binaire en C, on utilise un type faisant intervenir de la **récursivité** :

```c
typedef struct arb_bin {
  int val;
  struct arb_bin *gauche, *droite;
  // ^ pointeurs sur le type en train d'être défini
} arb_bin;
```

Attention, comme le type est récursif, il faut

- nommer une première fois le type `arb_bin` avant de le définir
- utiliser `struct` lorsque l'on déclare des pointeurs sur ce type dans la définition

Arbres binaire en C

Pour représenter un arbre binaire en C, on utilise un type faisant intervenir de la **récursivité** :

```c
typedef struct arb_bin {
  int val;
  struct arb_bin *gauche, *droite;
  // ^ pointeurs sur le type en train d'être défini
} arb_bin;
```

Par convention, on représente l'arbre vide par `NULL`.

Type impossible

Attention, le type suivant ne convient pas pour représenter les arbres binaires :

```
typedef struct arbbin {
  int val;
  struct arbbin left, right;
} arbbin;
```

Type impossible

Attention, le type suivant ne convient pas pour représenter les arbres binaires :

```
typedef struct arbbin {
  int val;
  struct arbbin left, right;
} arbbin;
```

Déjà, on n'a pas de moyens pour dire que le sous-arbre gauche ou droit est vide...

Type impossible

Attention, le type suivant ne convient pas pour représenter les arbres binaires :

```
typedef struct arbbin {
  int val;
  struct arbbin left, right;
} arbbin;
```

... mais de toute façon, `gcc` refuse ce type. En effet, on a

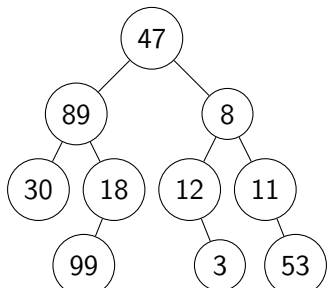
```
sizeof(arb_bin) = sizeof(int) + 2 * sizeof(arb_bin)
```

c'est-à-dire `sizeof(arb_bin)` $= +\infty$, donc ce type n'est pas instantiable.

Taille d'un arbre

La **taille d'un arbre** est le nombre de nœuds qui le constituent.

Exemple :



L'arbre ci-dessus a 10 nœuds.

Taille d'un arbre

```c
typedef struct arb_bin {
  int val;
  struct arb_bin *gauche, *droite;
} arb_bin;
```

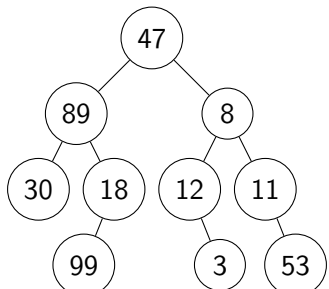
Calcul de la taille d'un arbre pour ce type :

```c
int calcul_taille(arb_bin *arb)
{
  if(arb == NULL)
    return 0;
  return 1 + calcul_taille(arb->gauche) + calcul_taille(arb->droite);
}
```

Profondeur d'un arbre

Rappel : la **profondeur** d'un arbre est la hauteur de son nœud le plus profond.

Exemple :



Les nœuds les plus hauts de cet arbre sont (99), (3) et (53), qui sont de hauteur 3.

La profondeur de l'arbre est donc 3.

Profondeur d'un arbre

```c
typedef struct arb_bin {
  int val;
  struct arb_bin *gauche, *droite;
} arb_bin;
```

Calcul de la profondeur d'un arbre pour ce type :

```c
int max(int a, int b){ return a < b ? b : a; }

int calcul_prof(arb_bin *arb)
{
  if(arb == NULL)
    return -1;
  return 1 + max(calcul_prof(arb->gauche),calcul_prof(arb->droite));
}
```

Création d'arbres

```
typedef struct arb_bin {
  int val;
  struct arb_bin *gauche, *droite;
} arb_bin;
```

Créer un arbre vide :

```
arb_bin* arb_vide()
{
  return NULL;
}
```

Création d'arbres

```
typedef struct arb_bin {
  int val;
  struct arb_bin *gauche, *droite;
} arb_bin;
```

Créer un arbre avec un seul nœud :

```
arb_bin* arb_singleton(int val)
{
  arb_bin *ptr = malloc(sizeof(arb_bin));
  ptr->val    = val;
  ptr->gauche = NULL;
  ptr->droite = NULL;
  return ptr;
}
```

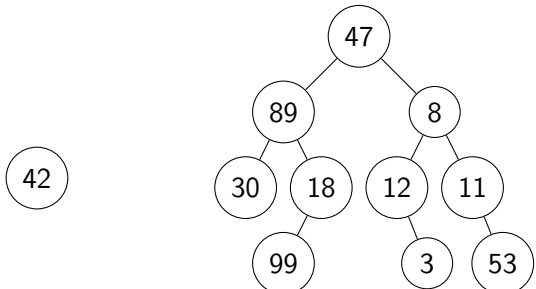
Destruction d'arbres

```c
typedef struct arb_bin {
  int val;
  struct arb_bin *gauche, *droite;
} arb_bin;
```

Détruire un arbre (récursivement) :

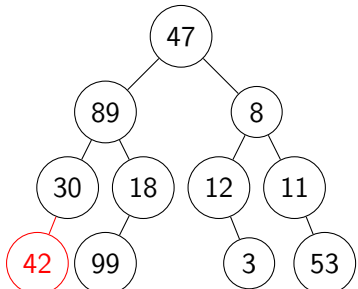
```c
void arb_detruire(arb_bin* arb)
{
  if(arb == NULL)
    return;
  arb_detruire(arb->gauche);
  arb_detruire(arb->droite);
  free(arb);
  return;
}
```

Insertion dans un arbre



On peut insérer un nœud à plusieurs endroits dans un arbre.

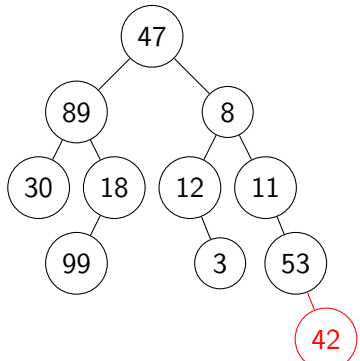
Insertion dans un arbre



On peut insérer un nœud à plusieurs endroits dans un arbre.

Par exemple, le plus à gauche possible.

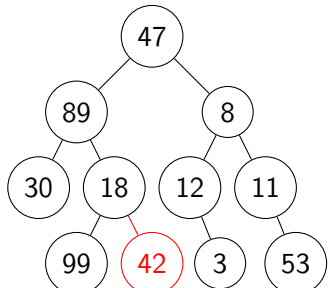
Insertion dans un arbre



On peut insérer un nœud à plusieurs endroits dans un arbre.

Par exemple, le plus à droite possible.

Insertion dans un arbre



On peut insérer un nœud à plusieurs endroits dans un arbre.

Par exemple, quelque part au « milieu ».

Insertion dans un arbre

```
typedef struct arb_bin {
  int val;
  struct arb_bin *gauche, *droite;
} arb_bin;
```

Insérer le plus à gauche possible :

```
void arb_inserer_gauche(arb_bin* arb, arb_bin *noeud) {
  if(arb->gauche == NULL)
    arb->gauche = noeud;
  else
    arb_inserer_gauche(arb->gauche,noeud);
  return;
}
```

Insertion dans un arbre

```
typedef struct arb_bin {
  int val;
  struct arb_bin *gauche, *droite;
} arb_bin;
```

Insérer le plus à droite possible :

```
void arb_inserer_droite(arb_bin* arb, arb_bin *noeud) {
  if(arb->droite == NULL)
    arb->droite = noeud;
  else
    arb_inserer_droite(arb->droite,noeud);
  return;
}
```

Insertion dans un arbre

```
typedef struct arb_bin {
  int val;
  struct arb_bin *gauche, *droite;
} arb_bin;
```

Insérer quelque part au hasard :

```
void arb_inserer_hasard(arb_bin* arb, arb_bin *noeud) {
  if((rand() % 2) == 0) {
    if(arb->gauche == NULL)
      arb->gauche = noeud;
    else
      arb_inserer_hasard(arb->gauche, noeud);
  }
  else {/* code symétrique qui insère à droite */ }
}
```

Insertion dans un arbre

```c
typedef struct arb_bin {
  int val;
  struct arb_bin *gauche, *droite;
} arb_bin;
```

Insérer quelque part au hasard (version concise avec double pointeur) :

```c
void arb_inserer_hasard(arb_bin* arb, arb_bin *noeud) {
  arb_bin **ptr = (rand() % 2) ? &arb->gauche : &arb->droite;
  if(*ptr == NULL)
    *ptr = noeud;
  else
    arb_inserer_hasard(*ptr,noeud);
}
```

Insertion dans un arbre

```
typedef struct arb_bin {
  int val;
  struct arb_bin *gauche, *droite;
} arb_bin;
```

En fait, les trois fonctions précédentes peuvent être utilisées pour insérer des sous-arbres entiers.

Il suffit de les invoquer en prenant pour `noeud` un pointeur sur un arbre.

Problème

Les fonctions que l'on a décrites ne permettent pas d'insérer dans un arbre vide.

```
void arb_inserer_gauche(arb_bin* arb, arb_bin *noeud) {
  if(arb->gauche == NULL)
    arb->gauche = noeud;
  else
    arb_inserer_gauche(arb->gauche,noeud);
  return;
}
```

Si `arb` est `NULL`, alors on fait un `segfault`.

De plus, à cause de l'appel par valeur, le pointeur de l'appelant ne sera pas changé.

Problème

Si l'on veut pouvoir insérer dans un arbre vide, on a plusieurs solutions.

Problème

Si l'on veut pouvoir insérer dans un arbre vide, on a plusieurs solutions.

Solution 1 : renvoyer un pointeur représentant le nouvel arbre.

```
arb_bin* arb_inserer_gauche(arb_bin* arb, arb_bin *noeud) {
  if(arb == NULL)
    return noeud;
  if(arb->gauche)
    arb->gauche = noeud;
  else
    arb_inserer_gauche(arb->gauche,noeud);
  return arb;
}
// ...
arb_bin *arb = ..., *noeud = ...;
arb = arb_inserer_gauche(arb,noeud); // réaffectation après appel
```

Problème

Si l'on veut pouvoir insérer dans un arbre vide, on a plusieurs solutions.

Solution 2 : prendre en argument un double pointeur.

```
void arb_inserer_gauche(arb_bin **arb_ptr, arb_bin *noeud) {
  if(*arb_ptr == NULL)
    *arb_ptr = noeud;
  else if((*arb_ptr)->gauche == NULL)
    (*arb_ptr)->gauche = noeud;
  else
    arb_inserer_gauche(&(*arb_ptr)->gauche,noeud);
  return;
}
// ...
arb_bin *arb = ..., *noeud = ...;
arb_inserer_gauche(&arb,noeud);
```

Problème

Sinon, on peut assumer et indiquer que ces fonctions ne traitent pas le cas de l'arbre vide.

```
void arb_inserer_gauche(arb_bin* arb, arb_bin *noeud) {
  assert(arb != NULL);
  // ...
}
```

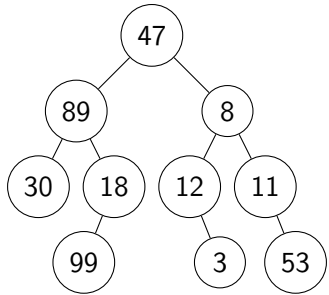
Arbre binaire de recherche

Soit T un arbre binaire.

T est un arbre binaire de recherche (*Binary Search Tree*) si chaque nœud de l'arbre vérifie que :

- tous les nœuds de son arbre gauche ont une valeur plus petite que lui ;
- tous les nœuds de son arbre droit ont une valeur plus grande que lui.

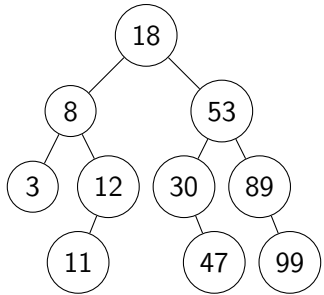
Exemple



L'arbre ci-dessus **n'est pas** un arbre binaire de recherche car :

- $47 < 89$ pourtant (89) est dans l'arbre gauche de (47)
- $18 < 89$ pourtant (18) est dans l'arbre droit de (89)
- *etc.*

Exemple



L'arbre ci-dessus **est** un arbre binaire de recherche car :

- $8 < 18$ et (8) est dans l'arbre gauche de (18)
- $53 < 89$ et (89) est dans l'arbre droit de (53)
- *etc.*

Insertion

Pour préserver la propriété des arbres binaires de recherche, on ne peut plus insérer directement des sous-arbres. La fonction d'insertion prend donc plutôt en argument la valeur à ajouter à l'arbre.

```
void arb_rech_inserer(arb_bin *arb, int v) {
  if(v < arb->val) {
    if(arb->gauche == NULL)
      arb->gauche = arb_singleton(v);
    else
      arb_rech_inserer(arb->gauche,v);
  }
  else if(v > arb->val){ /* cas symétrique */ }
  else return; // cas v == arb->val
  // ^ par convention, on ne rajoute pas un nœud qui existe déjà
}
```

Complexité : $O(\text{hauteur}(\texttt{arb}))$.

Recherche

La recherche consiste à déterminer si un élément est dans l'arbre.

La fonction suivante renvoie `1` si et seulement si `v` est dans `arb`.

```
int arb_rech_trouver(arb_bin *arb, int v)
{
  if(arb == NULL)
    return 0;
  if(v == arb->val)
    return 1;
  if(v <  arb->val)
    return arb_rech_trouver(arb->gauche,v);
  else
    return arb_rech_trouver(arb->droite,v);
}
```

Complexité : O(hauteur(`arb`)).

Intérêt

En équilibrant les différentes branches d'un arbre binaire, on peut faire tenir N nœuds dans un arbre de hauteur $O(\log(N))$.

Ainsi, on a une structure avec laquelle :

- on peut insérer des élément en $O(\log(N))$,
- on peut savoir si une valeur est présente en $O(\log(N))$

(on peut rééquilibrer l'arbre en $O(\log(N))$ après chaque insertion si nécessaire mais c'est technique à décrire)

Exercice : quelles seraient les complexités obtenues pour ces opérations en utilisant des tableaux ?

Sommaire

Définitions et exemples

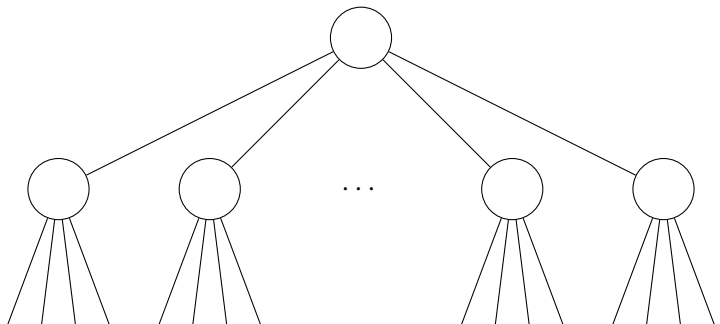
Arbres binaires

Arbres généraux

Nœuds de types différents

Types de parcours

Arbres avec plusieurs nœud enfants



Arbres avec plusieurs nœud enfants

On va adapter le code des arbres binaires pour contenir plus de nœuds enfants.

```
typedef struct arb_bin {
  int val;
  struct arb_bin *gauche, *droite;
} arb_bin;
```

Arbres avec plusieurs nœud enfants

On va adapter le code des arbres binaires pour contenir plus de nœuds enfants.

```
typedef struct arb_tab {
  int val;
  int n_enfants;
  struct arb_tab **enfants;
} arb_tab;
```

Par exemple, en ayant un pointeur `enfants` vers un **tableau** de pointeurs de nœuds.

On stocke de plus la taille de ce tableau dans `n_enfants`.

Arbres avec plusieurs nœud enfants

On va adapter le code des arbres binaires pour contenir plus de nœuds enfants.

```c
typedef struct arb_tab {
  int val;
  int n_enfants;
  struct arb_tab **enfants;
} arb_tab;
```

```c
int calcul_taille(arb_tab *arb) {
  if(arb == NULL) return 0;
  int reponse = 1;
  for(int i = 0; i < n_enfants; i++)
    reponse += calcul_taille(arb->enfants[i])
  return reponse;
}
```

Arbres avec plusieurs nœud enfants

On va adapter le code des arbres binaires pour contenir plus de nœuds enfants.

```
typedef struct arb_tab {
  int val;
  int n_enfants;
  struct arb_tab **enfants;
} arb_tab;
```

Pour ajouter un enfant à un nœud, il faut :

- créer un nouveau tableau `arb_tab **enfants2 = malloc(...)`
- recopier le contenu de `arb->enfants` dans `enfants2` en ajoutant le nouvel enfant à la bonne place
- désallouer l'ancier tableau avec `free(arb->enfants)`
- changer le pointeur avec `arb->enfants = enfants2`

Arbres avec plusieurs nœud enfants

On va adapter le code des arbres binaires pour contenir plus de nœuds enfants.

```
typedef struct arb_tab {
  int val;
  int n_enfants;
  struct arb_tab **enfants;
} arb_tab;
```

Avantage : on accède rapidement au $i^{\text{ème}}$ enfant d'un nœud ($O(1)$).

Inconvénient : ajouter (ou supprimer) un enfant à un nœud est une opération assez lourde ($O($ `n_enfants` $)$).

Arbres avec plusieurs nœud enfants

On va adapter le code des arbres binaires pour contenir plus de nœuds enfants.

```c
typedef struct arb_lc {
  int val;
  struct chainon *tete;
} arb_lc;
```

```c
typedef struct chainon {
  struct arb_lc  *noeud;
  struct chainon *suivant;
} chainon;
```

Ou sinon, en utilisant une liste chaînée au lieu d'un tableau.

Arbres avec plusieurs nœud enfants

On va adapter le code des arbres binaires pour contenir plus de nœuds enfants.

```c
typedef struct arb_lc {
  int val;
  struct chainon *tete;
} arb_lc;
```

```c
typedef struct chainon {
  struct arb_lc  *noeud;
  struct chainon *suivant;
} chainon;
```

```c
int calcul_taille(arb_lc *arb) {
  if(arb == NULL) return 0;
  int reponse = 1;
  for(chainon *actu = arb->tete; actu != NULL; actu = actu->suivant)
    reponse += calcul_taille(actu->noeud);
  return reponse;
}
```

Arbres avec plusieurs nœud enfants

On va adapter le code des arbres binaires pour contenir plus de nœuds enfants.

```
typedef struct arb_lc {
  int val;
  struct chainon *tete;
} arb_lc;
```

```
typedef struct chainon {
  struct arb_lc  *noeud;
  struct chainon *suivant;
} chainon;
```

Pour ajouter un enfant à un nœud, on utilise la procédure que l'on a écrite pour les listes chaînées.

Arbres avec plusieurs nœud enfants

On va adapter le code des arbres binaires pour contenir plus de nœuds enfants.

```c
typedef struct arb_lc {
  int val;
  struct chainon *tete;
} arb_lc;
```

```c
typedef struct chainon {
  struct arb_lc  *noeud;
  struct chainon *suivant;
} chainon;
```

Avantage : ajouter un enfant à un nœud est rapide ($O(1)$).

Inconvénient : l'accès au $i^{\text{ème}}$ enfant d'un nœud est lourd (O(nombre d'enfants))

Arbres avec plusieurs nœud enfants

On va adapter le code des arbres binaires pour contenir plus de nœuds enfants.

```
typedef struct arb_fe {
  int val;
  struct arb_fe *frere;
  struct arb_fe *enfant;
} arb_fe;
```

On peut simplifier les deux structures précédentes en une seule. On utilise :

- un champ `frere` qui pointe sur le frère suivant de ce nœud par rapport au parent
- un champ `enfant` qui pointe sur le premier enfant du nœud

Arbres avec plusieurs nœud enfants

On va adapter le code des arbres binaires pour contenir plus de nœuds enfants.

```
typedef struct arb_fe {
  int val;
  struct arb_fe *frere;
  struct arb_fe *enfant;
} arb_fe;
```

```
int calcul_taille(arb_fe *arb) {
  if(arb == NULL) return 0;
  return 1 + calcul_taille(arb->frere) + calcul_taille(arb->enfant);
}
```

Schéma pour `arb_fe`

En partant d'un arbre comme celui-ci...

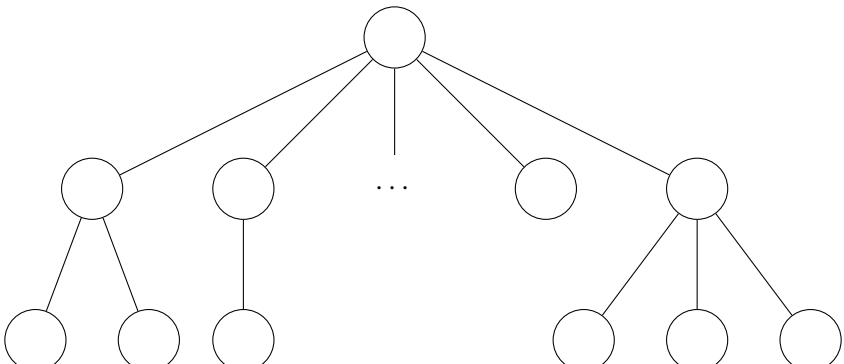
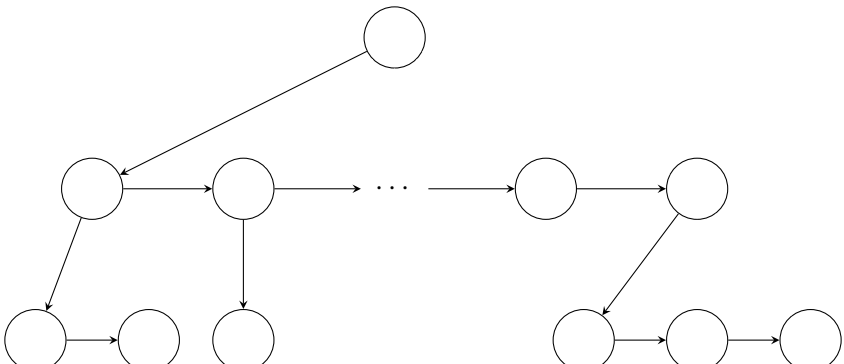


Schéma pour `arb_fe`

... on obtient les pointeurs `frere` et `enfant` suivants avec `arb_fe` :



Sommaire

Définitions et exemples

Arbres binaires

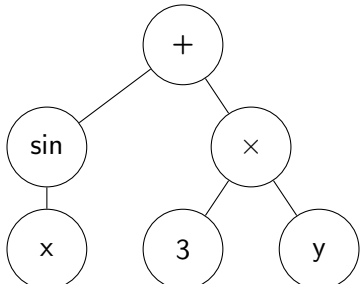
Arbres généraux

Nœuds de types différents

Types de parcours

Nœuds de types différents

On veut souvent manier des arbres qui contiennent des nœuds de types différents.



Ici, des nœuds différents :

- (x) et (y) sont des noms de variables
- (3) est une constante
- (+) et (×) sont des opérations binaires
- (sin) est une fonction à 1 argument

Plusieurs types en un seul

En C, on peut combiner plusieurs types en un seul en utilisant `stuct` :

```c
struct combinaison {
  int   entier;
  float flottant;
  char *chaine;
};
```

Cependant, chaque type est présent **en même temps** : dans une `combinaison`, on a

et un `entier` **et** un `flottant` **et** une `chaine`.

Dans notre cas, on souhaite plutôt **avoir le choix**, càd

soit un `entier`, **soit** un `flottant`, **soit** une `chaine`.

Plusieurs types en un seul

Pour n'avoir qu'un type à la fois, le C propose les `union` :

```
union choix {
  int   entier;
  float flottant;
  char *chaine;
};
```

Plusieurs types en un seul

Pour n'avoir qu'un type à la fois, le C propose les `union` :

```c
union choix {
  int   entier;
  float flottant;
  char *chaine;
};
```

On économise de la place par rapport aux `struct` car les données se chevauchent :

```c
union choix u;
u.entier = 42;
u.chaine = NULL;
printf("%d",u.entier); // affiche "0"
```

Plusieurs types en un seul

Pour n'avoir qu'un type à la fois, le C propose les `union` :

```
union choix {
  int   entier;
  float flottant;
  char *chaine;
};
```

En fait, on a :

`sizeof(struct combinaison)`

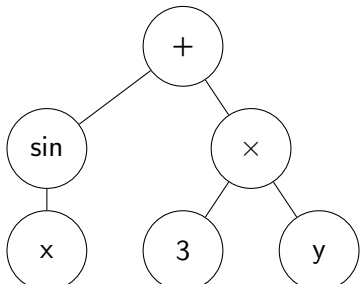
= `sizeof(int)` + `sizeof(float)` + `sizeof(char*)`

`sizeof(union choix)`

= max(`sizeof(int)` , `sizeof(float)` , `sizeof(char*)`)

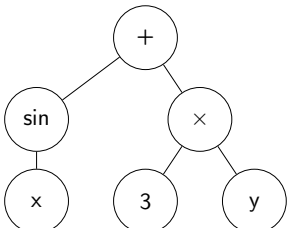
Plusieurs types en un seul

On peut ainsi utiliser des `union` pour les nœuds de notre exemple :



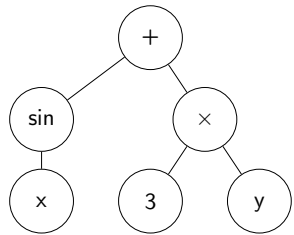
Plusieurs types en un seul

```
struct noeud {
  int type;
  union {
    char *variable;
    float constante;
    operateur_bin op;
    fonction fonc;
  } u;
};
```



Plusieurs types en un seul

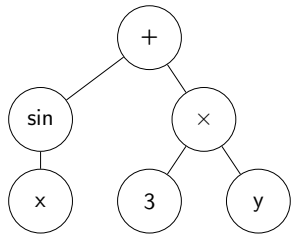
```
struct noeud {
  int type;
  union {
    char *variable;
    float constante;
    operateur_bin op;
    fonction fonc;
  } u;
};
```




```
typedef struct operateur_bin {
  int id;
  noeud *gauche, *droite;
} operateur_bin;
```

Plusieurs types en un seul

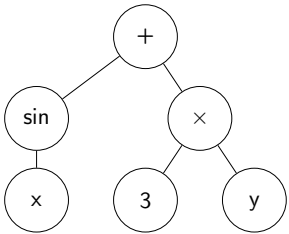
```
struct noeud {
  int type;
  union {
    char *variable;
    float constante;
    operateur_bin op;
    fonction fonc;
  } u;
};
```




```
typedef struct fonction {
  char *nom;
  noeud *argument;
} fonction;
```

Plusieurs types en un seul

```
struct noeud {
  int type;
  union {
    char *variable;
    float constante;
    operateur_bin op;
    fonction fonc;
  } u;
};
```



On prend une convention pour les types de nœuds :

- `type == 0` : une `variable`
- `type == 1` : une `constante`
- `type == 2` : un `op`
- `type == 3` : une `fonc`

Exemple : calcul de taille

```
int calcul_taille(noeud *node)
{
  if(node == NULL)
    return 0;
  if(  node->type == 0
    || node->type == 1) // cas d'une variable ou constante
    return 1;
  if(node->type == 2)   // cas d'un opérateur
    return 1 + calcul_taille(node->u.op.gauche)
             + calcul_taille(node->u.op.droite);
  if(node->type == 3)   // cas d'une fonction
    return 1 + calcul_taille(node->u.fonc.argument);
}
```

Problème

Supposons qu'on rajoute un autre type de nœud : un tableau de 100 `char`.

```
struct noeud {
  int type;
  union {
    char *variable;
    float constante;
    operateur_bin op;
    fonction fonc;
  } u;
};
```

Problème

Supposons qu'on rajoute un autre type de nœud : un tableau de 100 `char`.

```
struct noeud {
  int type;
  union {
    char *variable;
    float constante;
    operateur_bin op;
    fonction fonc;
    char tableau[100];  // <- ajouté
  } u;
};
```

Problème

Supposons qu'on rajoute un autre type de nœud : un tableau de 100 `char`.

```
struct noeud {
  int type;
  union {
    char *variable;
    float constante;
    operateur_bin op;
    fonction fonc;
    char tableau[100];  // <- ajouté
  } u;
};
```

- `type == 4` : un `tableau`

Problème

Supposons qu'on rajoute un autre type de nœud : un tableau de 100 `char`.

```
struct noeud {
  int type;
  union {
    char *variable;
    float constante;
    operateur_bin op;
    fonction fonc;
    char tableau[100];  // <- ajouté
  } u;
};
```

Maintenenant, on a

$$\texttt{sizeof(noeud)} = \texttt{sizeof(int)} + \max(\texttt{sizeof(char*)}, \ldots, \texttt{sizeof(char[100])})$$
$$= 4 + 100 = 104$$

Problème

Supposons qu'on rajoute un autre type de nœud : un tableau de 100 `char`.

```
struct noeud {
  int type;
  union {
    char *variable;
    float constante;
    operateur_bin op;
    fonction fonc;
    char tableau[100];  // <- ajouté
  } u;
};
```

Ainsi, même si un `noeud` de `type` ≤ 3 n'utilise que 12 octets au plus, sa taille sera quand même de 104 octets à cause de l'`union` et de `tableau` : **gâchis de mémoire**.

Solution avec `void*`

Pour ne pas avoir ce problème, on va représenter les nœuds d'une autre façon sans utiliser d'`union` :

```
struct noeud {
  int type;
  union {
    char *variable;
    float constante;
    operateur_bin op;
    fonction fonc;
    char tableau[100];
  } u;
};
```

Solution avec `void*`

Pour ne pas avoir ce problème, on va représenter les nœuds d'une autre façon sans utiliser d'`union` :

```
struct noeud {
  int type;
  void *donnees;
};
```

Solution avec `void*`

Pour ne pas avoir ce problème, on va représenter les nœuds d'une autre façon sans utiliser d'`union` :

```
struct noeud {
  int type;
  void *donnees;
};
```

`donnees` est un pointeur sur un des anciens types de l'`union`.

Le « vrai » type de `donnees` est donné par `type` :

- `type == 0` (cas `variable`) : `char**`
- `type == 1` (cas `constante`) : `float*`
- `type == 2` (cas `op`) : `operateur_bin*`
- *etc.*

Solution avec `void*`

Pour ne pas avoir ce problème, on va représenter les nœuds d'une autre façon sans utiliser d'`union` :

```
struct noeud {
  int type;
  void *donnees;
};
```

`donnees` est un pointeur sur un des anciens types de l'`union`.

Le « vrai » type de `donnees` est donné par `type` :

- `type == 0` (cas `variable`) : `char*` (on peut se passer de double pointeur ici)
- `type == 1` (cas `constante`) : `float*`
- `type == 2` (cas `op`) : `operateur_bin*`
- *etc.*

Solution avec `void*`

Pour ne pas avoir ce problème, on va représenter les nœuds d'une autre façon sans utiliser d'`union` :

```
struct noeud {
  int type;
  void *donnees;
};
```

Ainsi, il faudra convertir `donnees` selon `type` pour accéder au contenu du `noeud` :

```
void foo(noeud *node) {
  if(node->type == 0) {
    char *ptr = (char*) node->donnees;
    // on fait des choses avec ptr
  }
  // ...
}
```

Solution avec `void*`

Pour ne pas avoir ce problème, on va représenter les nœuds d'une autre façon sans utiliser d'`union` :

```
struct noeud {
  int type;
  void *donnees;
};
```

Ainsi, il faudra convertir `donnees` selon `type` pour accéder au contenu du `noeud` :

```
// ...
  else if(node->type == 1) {
    float *ptr = (float*) node->donnees;
    // on fait des choses avec ptr
  }
  // ...
}
```

Solution avec `void*`

Pour ne pas avoir ce problème, on va représenter les nœuds d'une autre façon sans utiliser d'`union` :

```
struct noeud {
  int type;
  void *donnees;
};
```

Ainsi, il faudra convertir `donnees` selon `type` pour accéder au contenu du `noeud` :

```
// ...
  else if(node->type == 2) {
    operateur_bin *ptr = (operateur_bin*) node->donnees;
    // on fait des choses avec ptr
  }
  // ...
}
```

Solution avec `void*`

Pour ne pas avoir ce problème, on va représenter les nœuds d'une autre façon sans utiliser d'`union` :

```
struct noeud {
  int type;
  void *donnees;
};
```

Ainsi, il faudra convertir `donnees` selon `type` pour accéder au contenu du `noeud` :

etc.

Exemple

On reprend notre définition de nœud à 4 types :

```
struct noeud {
  int type;
  void *donnees;
};
```

- `type == 0` : un `char*`
- `type == 1` : un `float`
- `type == 2` : un `operateur_bin`
- `type == 3` : une `fonction`

```
typedef struct operateur_bin {
  int id;
  noeud *gauche, *droite;
} operateur_bin;
```

```
typedef struct fonction {
  char *nom;
  noeud *argument;
} fonction;
```

Exemple

```
int calcul_taille(noeud *node)
{
  if(node == NULL)
    return 0;
  if(  node->type == 0
    || node->type == 1) // cas d'une variable ou constante
    return 1;
  if(node->type == 2){  // cas d'un opérateur
    operateur_bin *ptr = (operateur_bin*) node->donnees;
    return 1 + calcul_taille(ptr->gauche)
             + calcul_taille(ptr->droite);
  }
  if(node->type == 3){  // cas d'une fonction
    fonction *ptr = (fonction*) node->donnees;
    return 1 + calcul_taille(ptr->argument);
  }
  errx(1,"Type de nœud inconnu (%d)",node->type);
}
```

Sommaire

Définitions et exemples

Arbres binaires

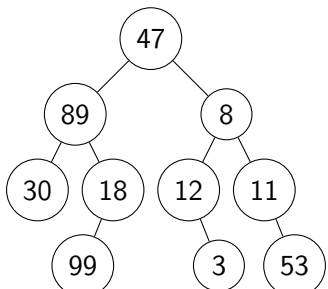
Arbres généraux

Nœuds de types différents

Types de parcours

Parcours sur les arbres binaires

Étant donné un arbre binaire



on souhaite souvent appliquer une action `f(v)` sur chacune des valeurs `v` de l'arbre.

Parcours sur les arbres binaires

Par exemple, on peut vouloir afficher chacune des valeurs en appliquant la fonction `afficher` à chaque valeur contenue dans l'arbre :

```
void afficher(int v)
{
  printf("%d ", v);
}
```

Parcours sur les arbres binaires

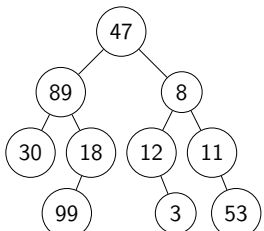
Par exemple, on peut vouloir afficher chacune des valeurs en appliquant la fonction `afficher` à chaque valeur contenue dans l'arbre :

```
void afficher(int v)
{
  printf("%d ", v);
}
```

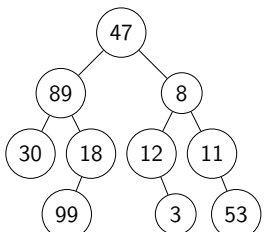
On a alors essentiellement 3 choix pour l'ordre de parcours des nœuds :

- **parcours préfixe**
- **parcours infixe**
- **parcours postfixe**

Parcours sur les arbres binaires



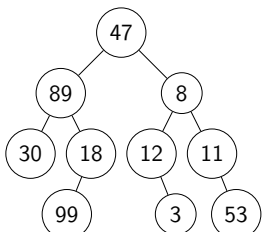
Parcours sur les arbres binaires



Parcours préfixe :

```
void afficher_prefixe(arb_bin *arb) {
  if(arb == NULL)
  return;
  afficher(arb);
  afficher_prefixe(arb->gauche);
  afficher_prefixe(arb->droite);
}
```

Parcours sur les arbres binaires

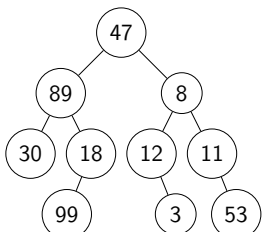


Parcours préfixe :

```
void afficher_prefixe(arb_bin *arb) {
  if(arb == NULL)
  return;
  afficher(arb);
  afficher_prefixe(arb->gauche);
  afficher_prefixe(arb->droite);
}
```

```
47 89 30 18 99 8 12 3 11 53
```

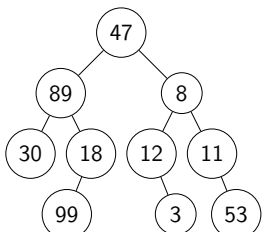
Parcours sur les arbres binaires



Parcours **infixe** :

```
void afficher_infixe(arb_bin *arb) {
  if(arb == NULL)
  return;
  afficher_infixe(arb->gauche);
  afficher(arb);                    // <-
  afficher_infixe(arb->droite);
}
```

Parcours sur les arbres binaires

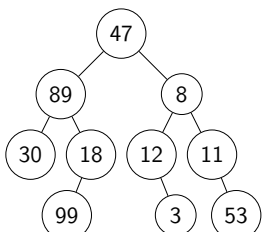


Parcours **infixe** :

```
void afficher_infixe(arb_bin *arb) {
  if(arb == NULL)
  return;
  afficher_infixe(arb->gauche);
  afficher(arb);                    // <-
  afficher_infixe(arb->droite);
}
```

```
30 89 99 18 47 12 3 8 11 53
```

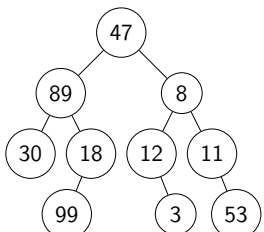
Parcours sur les arbres binaires



Parcours **postfixe** :

```
void afficher_postfixe(arb_bin *arb) {
  if(arb == NULL)
  return;
  afficher_postfixe(arb->gauche);
  afficher_postfixe(arb->droite);
  afficher(arb);                    // <-
}
```

Parcours sur les arbres binaires



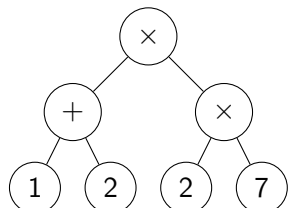
Parcours **postfixe** :

```
void afficher_postfixe(arb_bin *arb) {
  if(arb == NULL)
  return;
  afficher_postfixe(arb->gauche);
  afficher_postfixe(arb->droite);
  afficher(arb);                    // <-
}
```

```
30 99 18 89 3 12 53 11 8 47
```

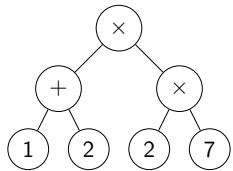
Arbres arithmétiques

$(1 + 2) \times (2 \times 7) \quad \rightsquigarrow$



Un cas intéressant de comparaison pour ces différentes formes de parcours est les **arbres d'expression arithmétiques**.

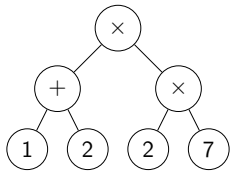
Arbres arithmétiques




```c
typedef struct arb_arith {
  int type;
  enum {
    int constante;
    struct operateur op;
  } u;
} arb_arith;
```

```c
typedef struct operateur {
  int type_op;
  arb_arith *gauche, *droite;
} operateur;
```

Arbres arithmétiques

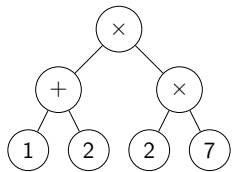



```
typedef struct arb_arith {
  int type;
  enum {
    int constante;
    struct operateur op;
  } u;
} arb_arith;
```

Type d'un nœud `arb_arith` :

- `type == 0` : `constante`
- `type == 1` : `op`

Arbres arithmétiques

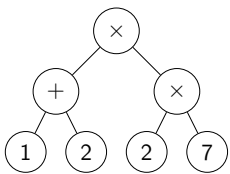


Type d'un `operateur` :

- `type_op == 0` : ×
- `type_op == 1` : +

```
typedef struct operateur {
  int type_op;
  arb_arith *gauche, *droite;
} operateur;
```

Arbres arithmétiques

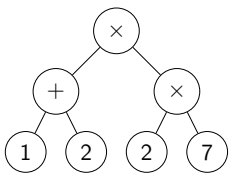


L'affichage standard s'obtient avec du parcours infixe :

```
void afficher_arith_infixe(arb_arith *arb) {
  if(arb->type == 0)
  printf("%d",arb->u.constante);
  else { // arb->type == 1
    afficher_arith_infixe(arb->u.op.gauche);
    printf(" %c ",arb->u.op.type_op ? '+', '*');
    afficher_arith_infixe(arb->u.op.droite);
  }
}
```

```
1 + 2 * 2 * 7
```

Arbres arithmétiques



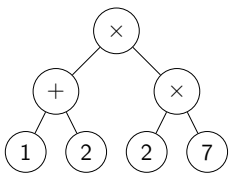
L'affichage standard s'obtient avec du parcours infixe :

```
void afficher_arith_infixe(arb_arith *arb) {
  if(arb->type == 0)
  printf("%d",arb->u.constante);
  else { // arb->type == 1
    afficher_arith_infixe(arb->u.op.gauche);
    printf(" %c ",arb->u.op.type_op ? '+', '*');
    afficher_arith_infixe(arb->u.op.droite);
  }
}
```

```
1 + 2 * 2 * 7
```

Ou presque : il faut **rajouter les parenthèses** !

Arbres arithmétiques

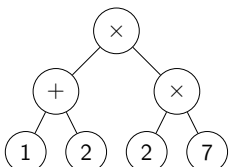


L'affichage standard s'obtient avec du parcours infixe :

```
void afficher_arith_infixe(arb_arith *arb) {
  if(arb->type == 0)
  printf("%d",arb->u.constante);
  else { // arb->type == 1
    printf("(");
    afficher_arith_infixe(arb->u.op.gauche);
    printf(" %c ",arb->u.op.type_op ? '+', '*');
    afficher_arith_infixe(arb->u.op.droite);
    printf(")");
  }
}
```

```
((1 + 2) * (2 * 7))
```

Arbres arithmétiques

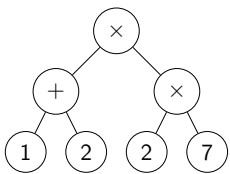


Avec l'affichage préfixe on obtient la **notation polonaise** :

```
void afficher_arith_prefixe(arb_arith *arb) {
  if(arb->type == 0)
  printf("%d ",arb->u.constante);
  else { // arb->type == 1
    printf("%c ",arb->u.op.type_op ? '+', '*');
    afficher_arith_prefixe(arb->u.op.gauche);
    afficher_arith_prefixe(arb->u.op.droite);
  }
}
```

```
* + 1 2 * 2 7
```

Arbres arithmétiques



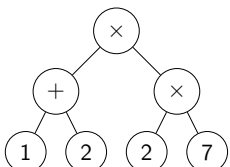
Avec l'affichage préfixe on obtient la **notation polonaise** :

```
void afficher_arith_prefixe(arb_arith *arb) {
  if(arb->type == 0)
  printf("%d ",arb->u.constante);
  else { // arb->type == 1
    printf("%c ",arb->u.op.type_op ? '+', '*');
    afficher_arith_prefixe(arb->u.op.gauche);
    afficher_arith_prefixe(arb->u.op.droite);
  }
}
```

```
* + 1 2 * 2 7
```

On n'a **pas besoin de parenthèses** avec cette notation !

Arbres arithmétiques

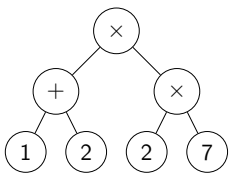


En postfixe, on obtient la **notation polonaise inverse** :

```
void afficher_arith_postfixe(arb_arith *arb) {
  if(arb->type == 0)
  printf("%d ",arb->u.constante);
  else { // arb->type == 1
    afficher_arith_postfixe(arb->u.op.gauche);
    afficher_arith_postfixe(arb->u.op.droite);
    printf("%c ",arb->u.op.type_op ? '+', '*');
  }
}
```

```
1 2 + 2 7 * *
```

Arbres arithmétiques



En postfixe, on obtient la **notation polonaise inverse** :

```
void afficher_arith_postfixe(arb_arith *arb) {
  if(arb->type == 0)
  printf("%d ",arb->u.constante);
  else { // arb->type == 1
    afficher_arith_postfixe(arb->u.op.gauche);
    afficher_arith_postfixe(arb->u.op.droite);
    printf("%c ",arb->u.op.type_op ? '+', '*');
  }
}
```

```
1 2 + 2 7 * *
```

À nouveau, **pas besoin de parenthèses** !