Partie 2: Arbres

Marie-Pierre Béal, Laurent Bulteau

Université Paris-Est Marne-la-Vallée

Plan

Exemples

Structures non-linéaires

- Arbres généalogiques
- Systèmes de fichiers
- Documents structurés
- Autres...

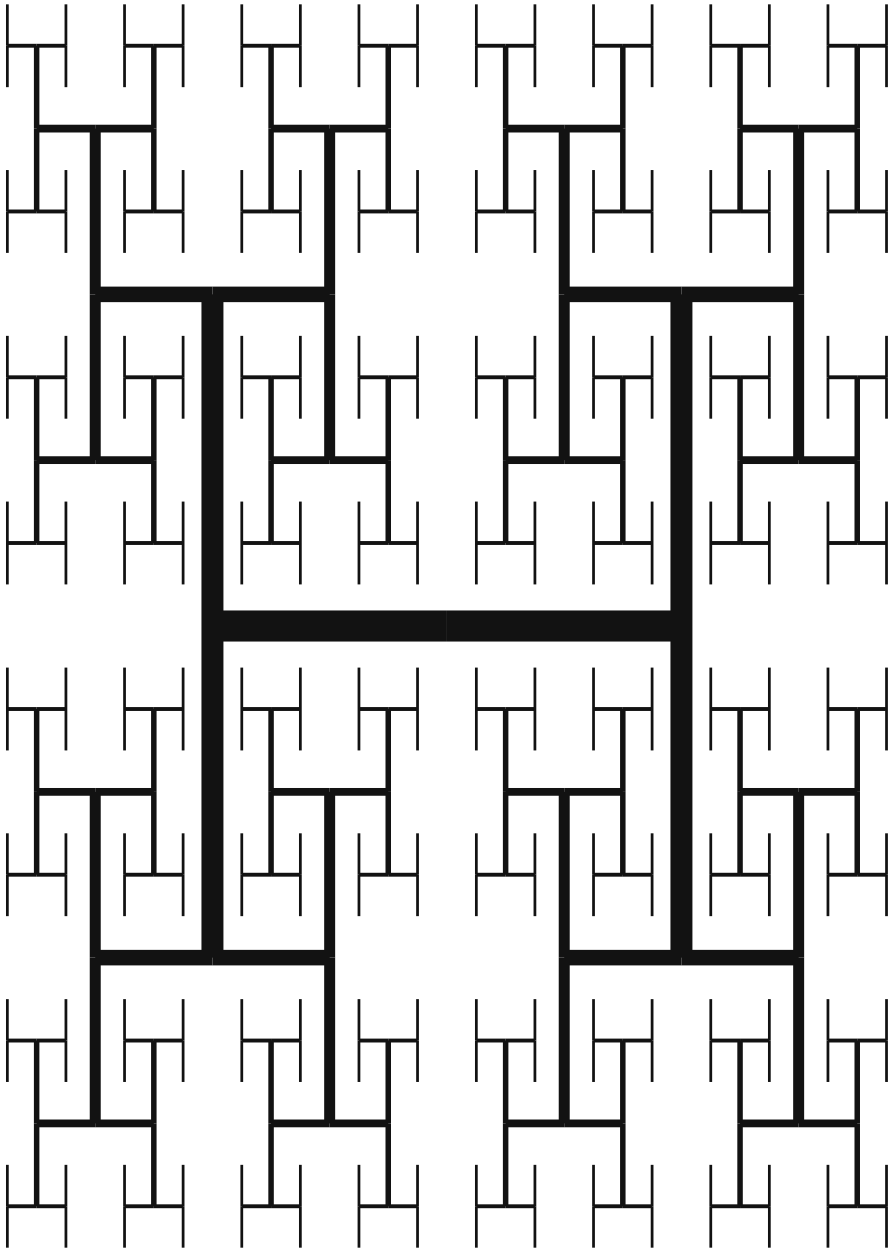
Exemples

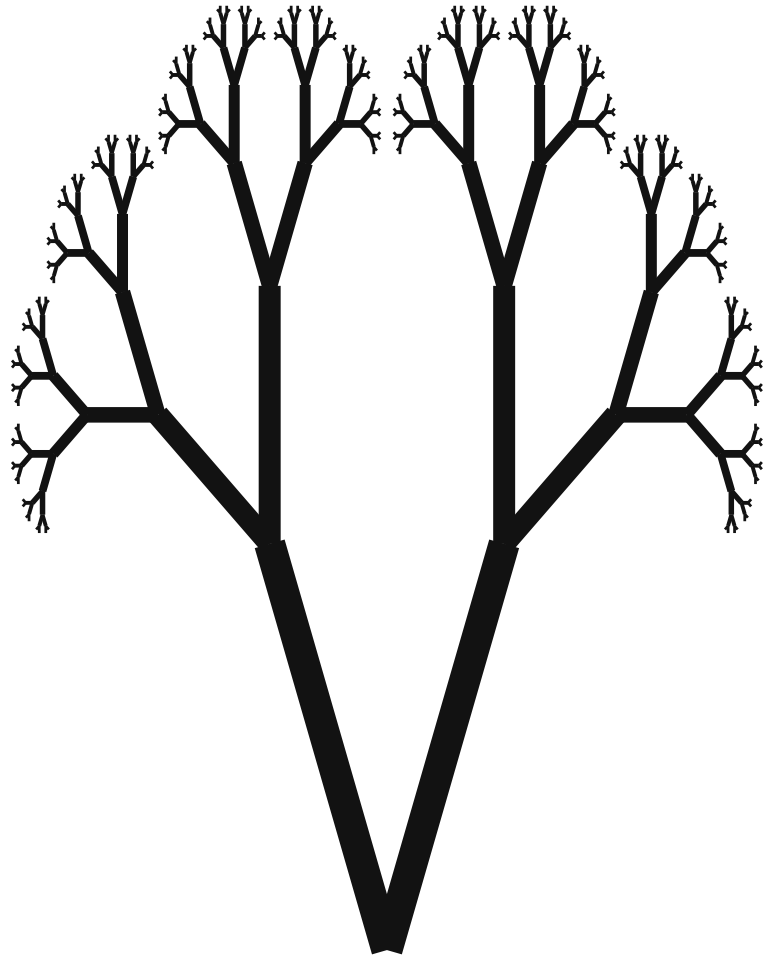
Structures non-linéaires

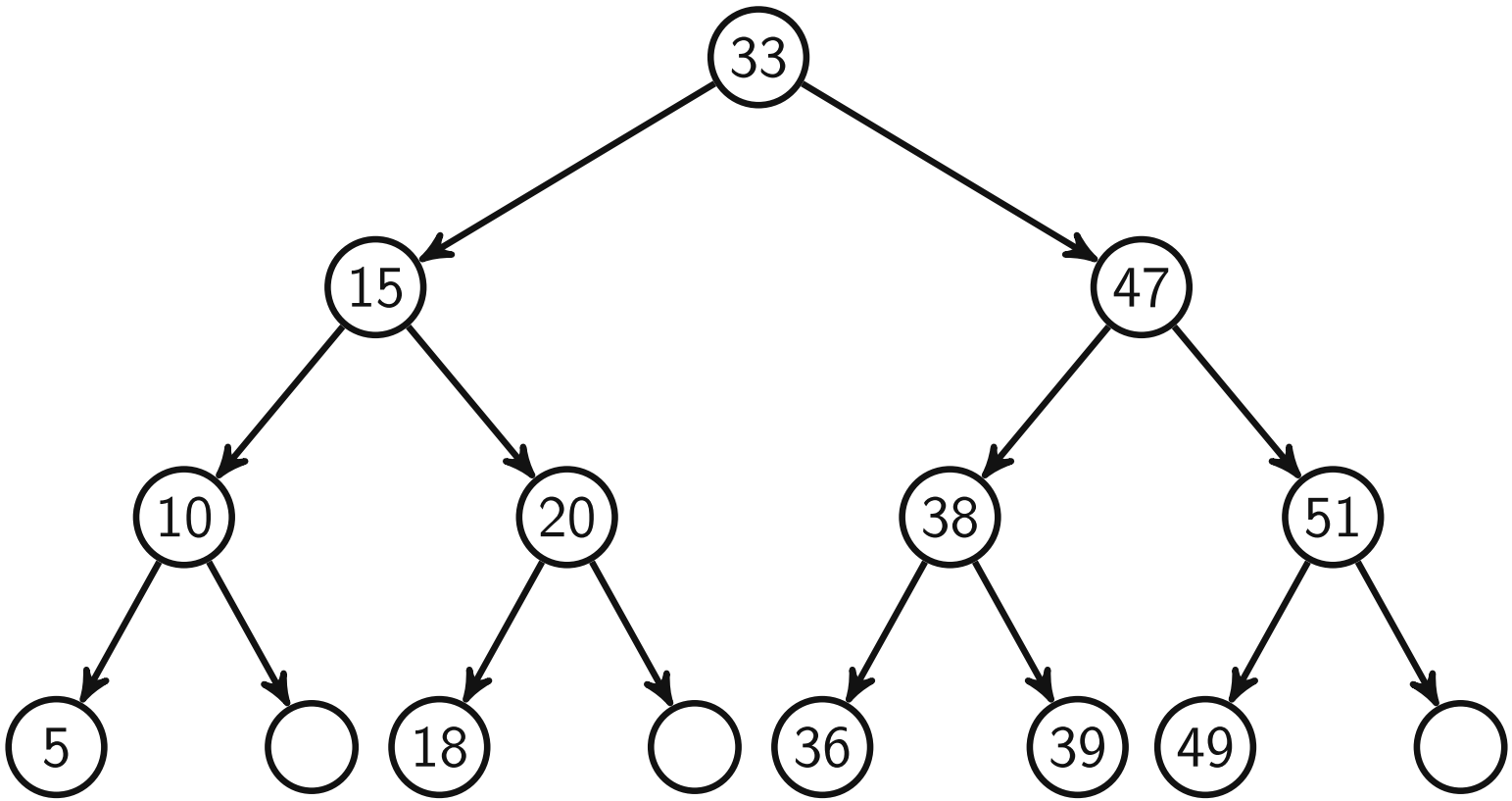
- Arbres généalogiques
- Systèmes de fichiers
- Documents structurés
- Autres...

XML

Extensible Markup Language, langage de balisage extensible (en Français), est un langage de balisage *générique* héritier de SGML. La syntaxe est dite « extensible » car elle permet de définir des espaces de noms, c'est-à-dire des dialectes avec chacun leur vocabulaire et leur grammaire.
Exemple : XHTML, XSLT, RSS, SVG.






33
15
47
10
20
38
51
5
18
36
39
49

Définitions

Définition récursive

Un arbre est soit *l'arbre vide*, soit composé de :

- Un nœud appelé *racine* (r).
- Un nombre arbitraire d'arbres (non-vides) appelés *sous-arbres*.

Définitions

Définition récursive

Un arbre est soit *l'arbre vide*, soit composé de :

- Un nœud appelé *racine* (r).
  - Ce nœud contient généralement une *étiquette* d'un type T donné
- Un nombre arbitraire d'arbres (non-vides) appelés *sous-arbres*.

Définitions

Définition récursive

Un arbre est soit *l'arbre vide*, soit composé de :

- Un nœud appelé *racine* (r).
  - Ce nœud contient généralement une *étiquette* d'un type T donné
- Un nombre arbitraire d'arbres (non-vides) appelés *sous-arbres*.
  - Chaque sous-arbre a sa propre racine : ce sont les *fils* de r (r est leur *père*)

Définitions

Définition récursive

Un arbre est soit *l'arbre vide*, soit composé de :

- Un nœud appelé *racine* (r).
  - Ce nœud contient généralement une *étiquette* d'un type T donné
- Un nombre arbitraire d'arbres (non-vides) appelés *sous-arbres*.
  - Chaque sous-arbre a sa propre racine : ce sont les *fils* de r (r est leur *père*)

 :

Définitions

Définition récursive

Un arbre est soit *l'arbre vide*, soit composé de :

- Un nœud appelé *racine* (r).
  - Ce nœud contient généralement une *étiquette* d'un type T donné
- Un nombre arbitraire d'arbres (non-vides) appelés *sous-arbres*.
  - Chaque sous-arbre a sa propre racine : ce sont les *fils* de r (r est leur *père*)

 : $\emptyset$

Définitions

Définition récursive

Un arbre est soit *l'arbre vide*, soit composé de :

- Un nœud appelé *racine* (r).
  - Ce nœud contient généralement une *étiquette* d'un type T donné
- Un nombre arbitraire d'arbres (non-vides) appelés *sous-arbres*.
  - Chaque sous-arbre a sa propre racine : ce sont les *fils* de r (r est leur *père*)

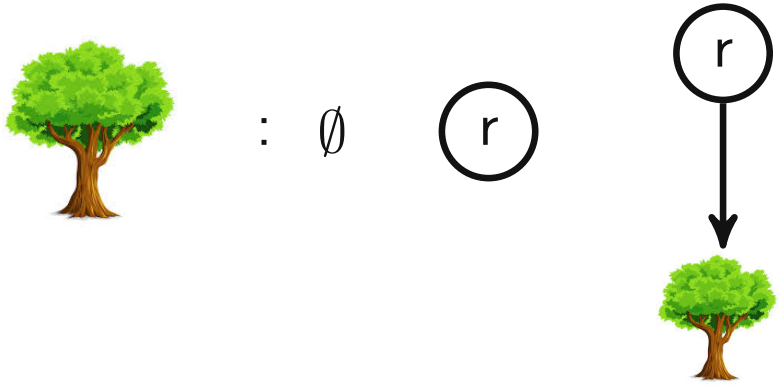
: $\emptyset$ 

Définitions

Définition récursive

Un arbre est soit *l'arbre vide*, soit composé de :

- Un nœud appelé *racine* (r).
  - Ce nœud contient généralement une *étiquette* d'un type T donné
- Un nombre arbitraire d'arbres (non-vides) appelés *sous-arbres*.
  - Chaque sous-arbre a sa propre racine : ce sont les *fils* de r (r est leur *père*)

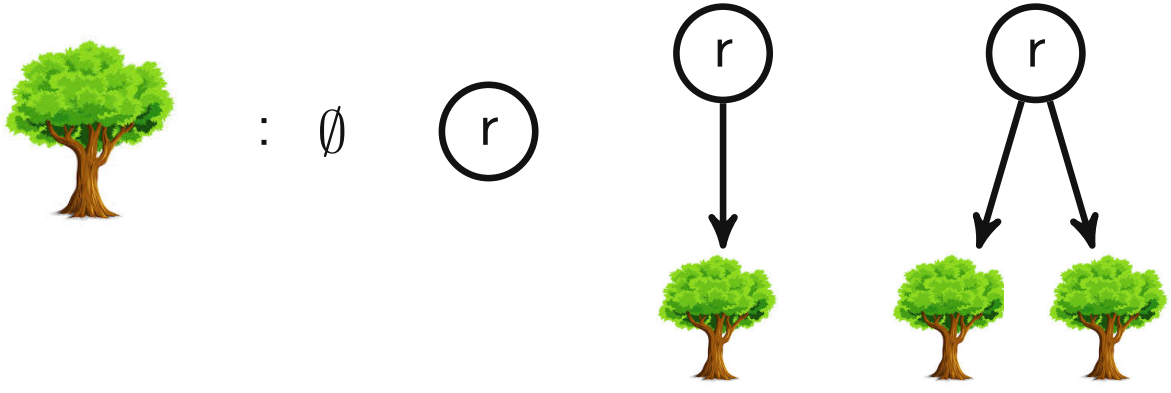


Définitions

Définition récursive

Un arbre est soit *l'arbre vide*, soit composé de :

- Un nœud appelé *racine* (r).
  - Ce nœud contient généralement une *étiquette* d'un type T donné
- Un nombre arbitraire d'arbres (non-vides) appelés *sous-arbres*.
  - Chaque sous-arbre a sa propre racine : ce sont les *fils* de r (r est leur *père*)

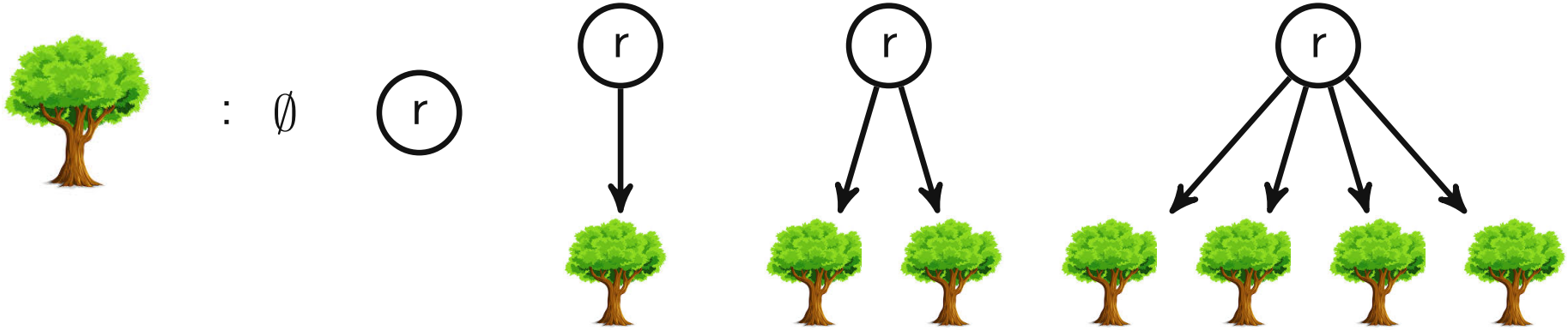


Définitions

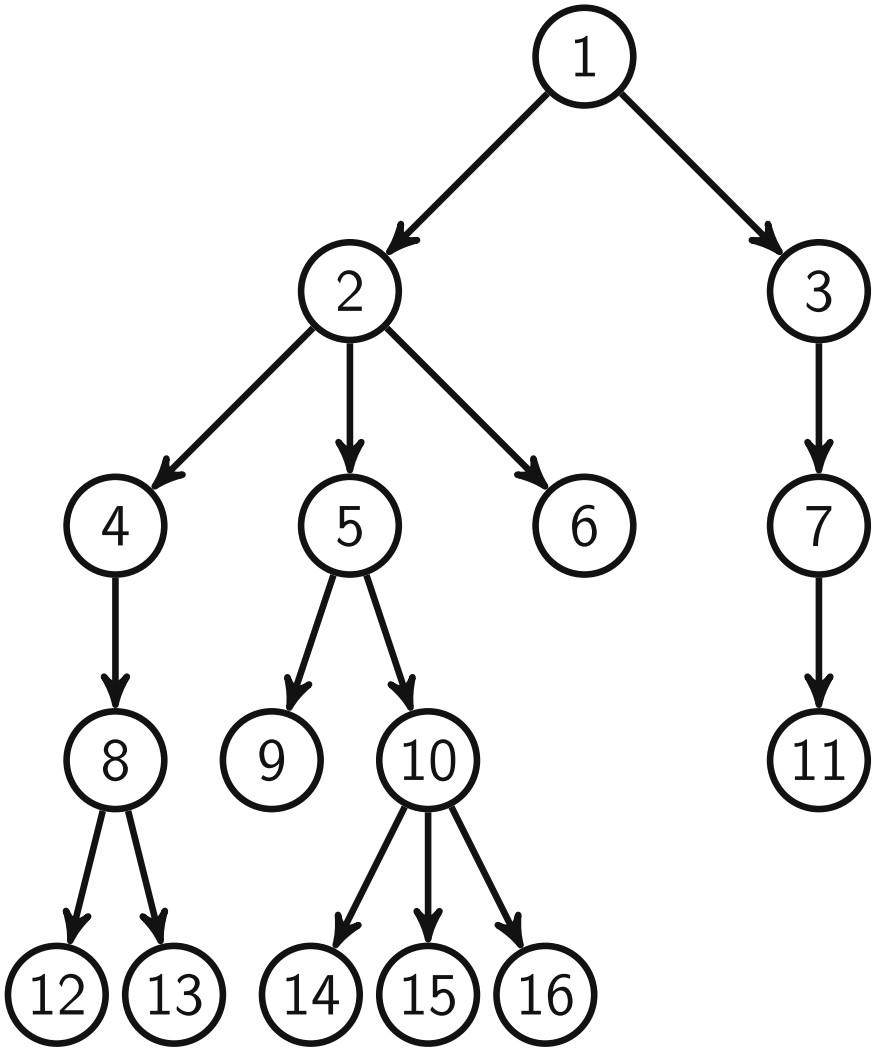
Définition récursive

Un arbre est soit *l'arbre vide*, soit composé de :

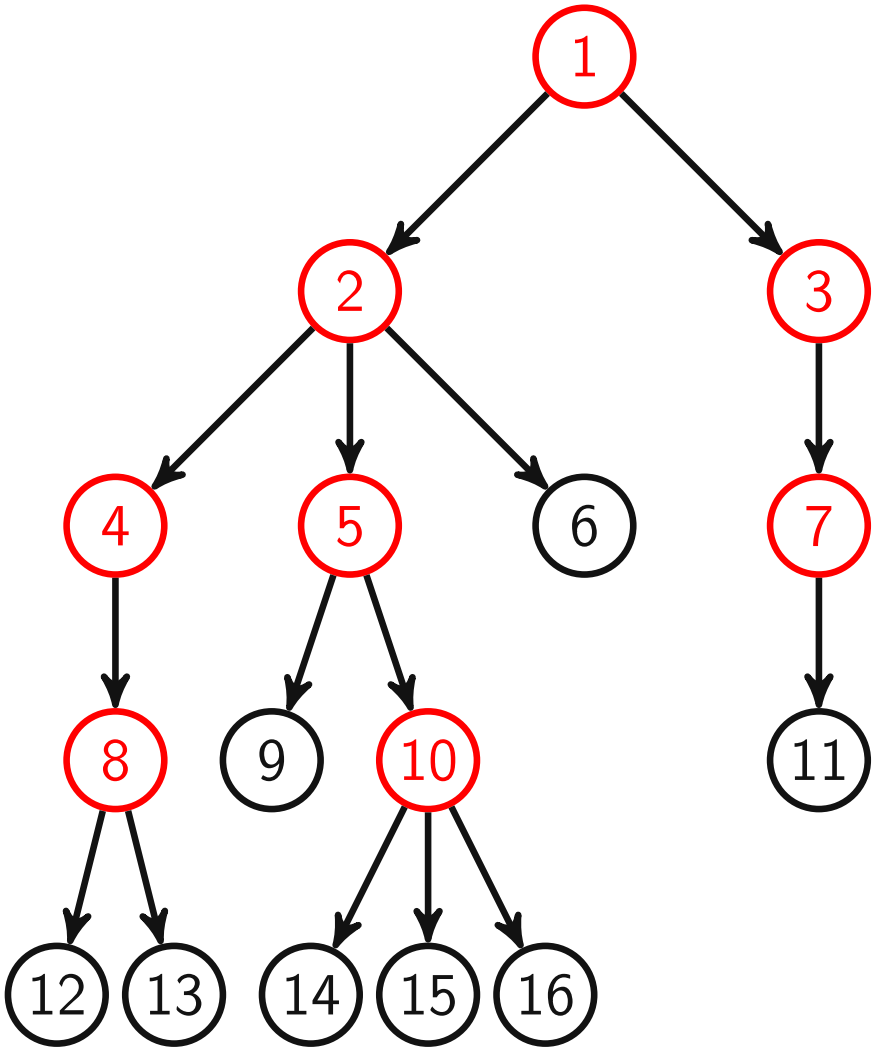
- Un nœud appelé *racine* (r).
  - Ce nœud contient généralement une *étiquette* d'un type T donné
- Un nombre arbitraire d'arbres (non-vides) appelés *sous-arbres*.
  - Chaque sous-arbre a sa propre racine : ce sont les *fils* de r (r est leur *père*)



Définitions

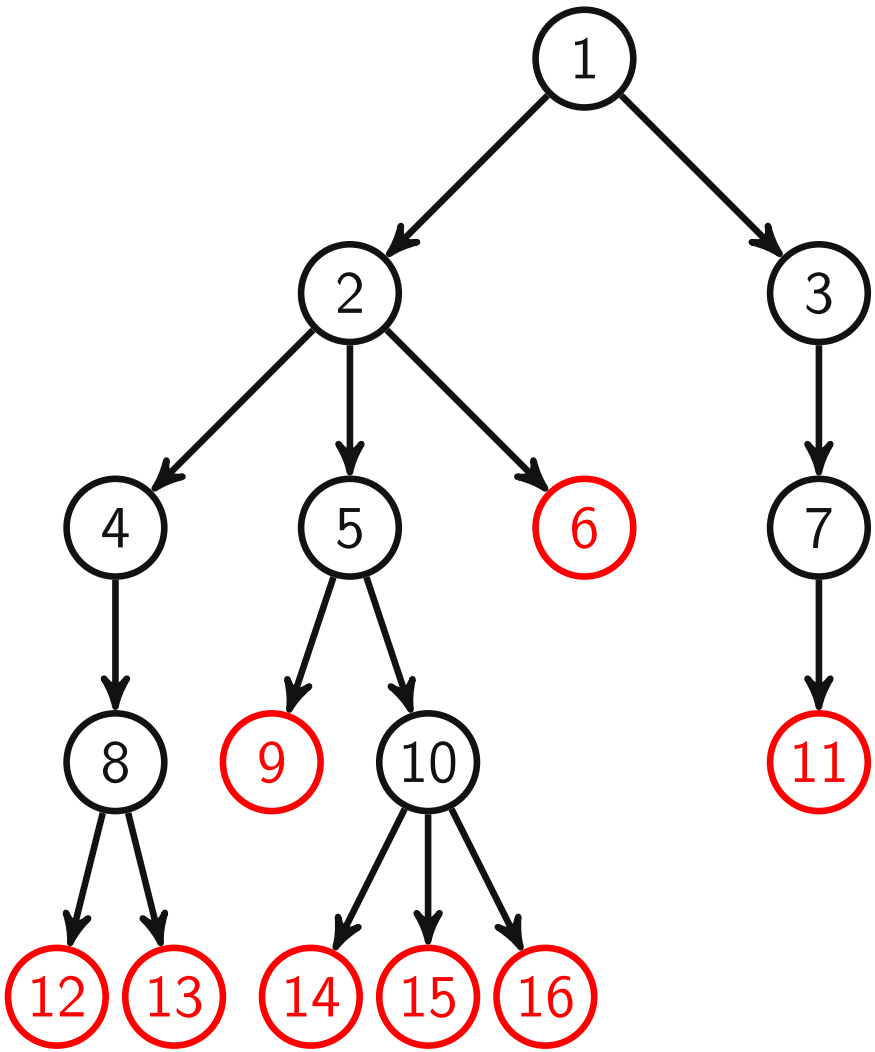


Définitions



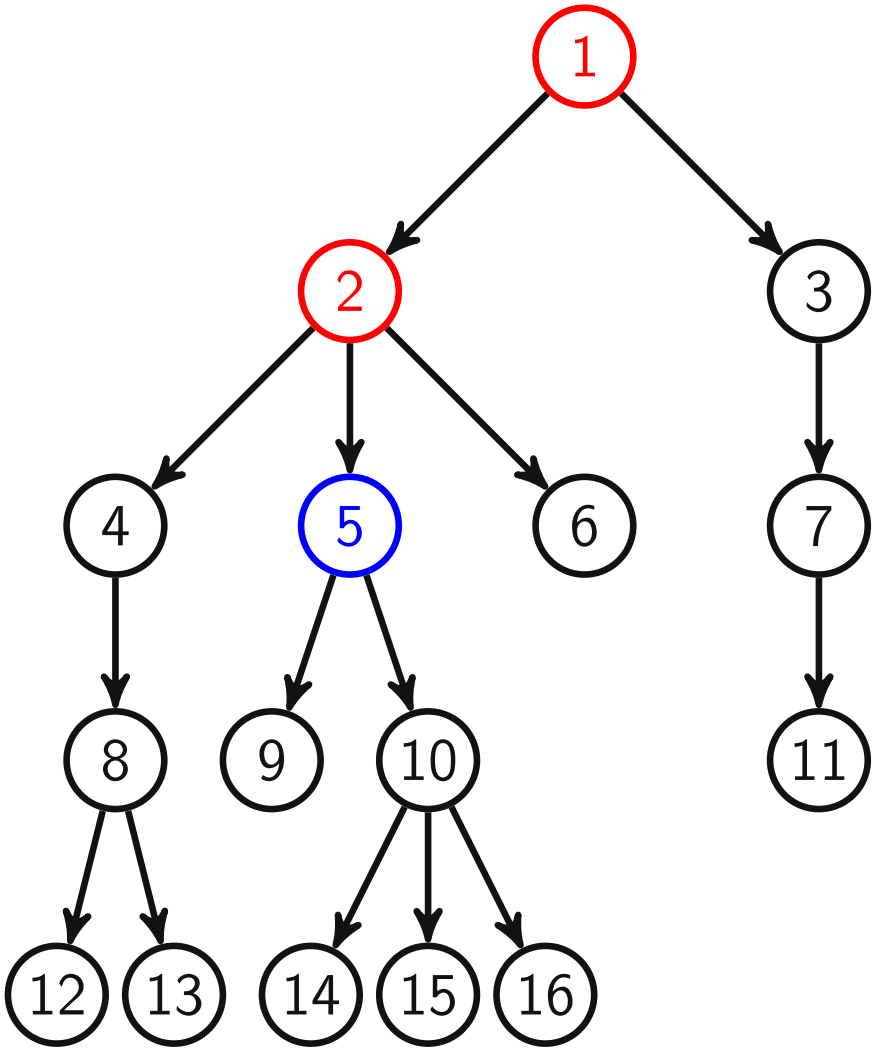
- *Nœud interne* : Un nœud qui possède au moins un fils ;

Définitions



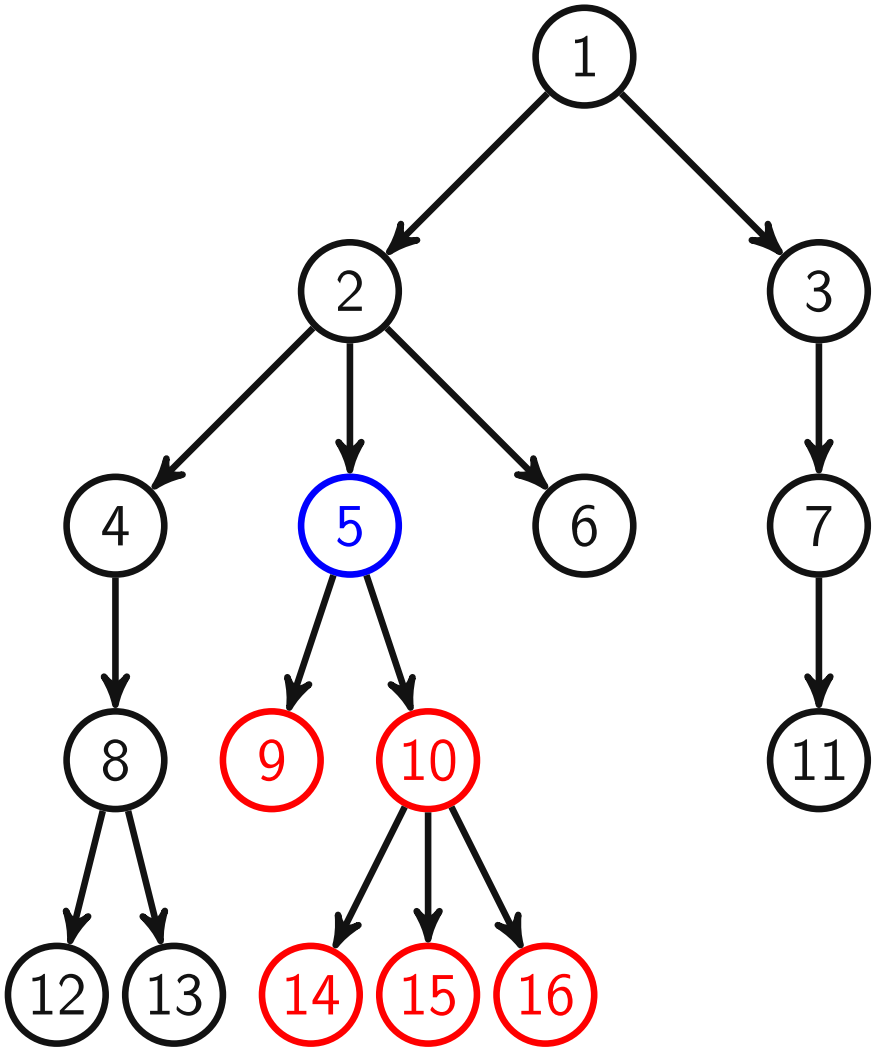
- *Nœud interne* : Un nœud qui possède au moins un fils ;
- *Feuille* : Un nœud sans aucun fils ;

Définitions



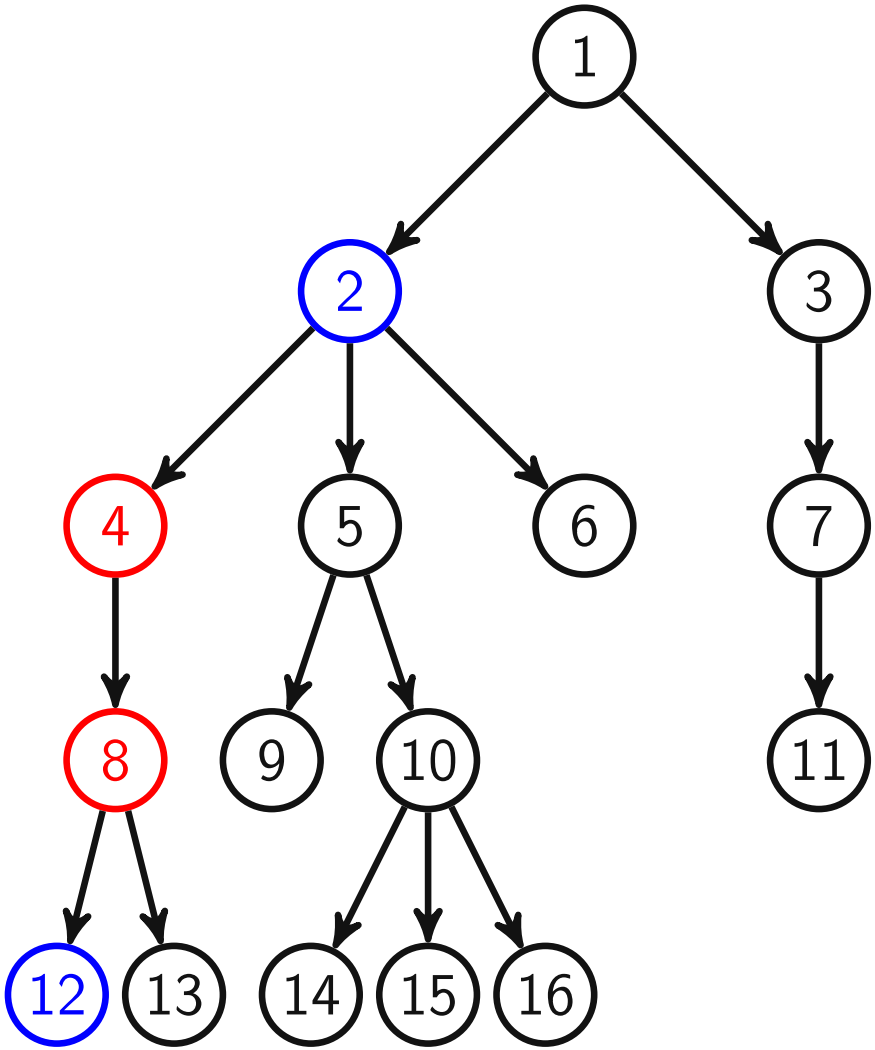
- *Nœud interne* : Un nœud qui possède au moins un fils ;
- *Feuille* : Un nœud sans aucun fils ;
- *Ascendants* : Le père d'un nœud et ses ascendants

Définitions



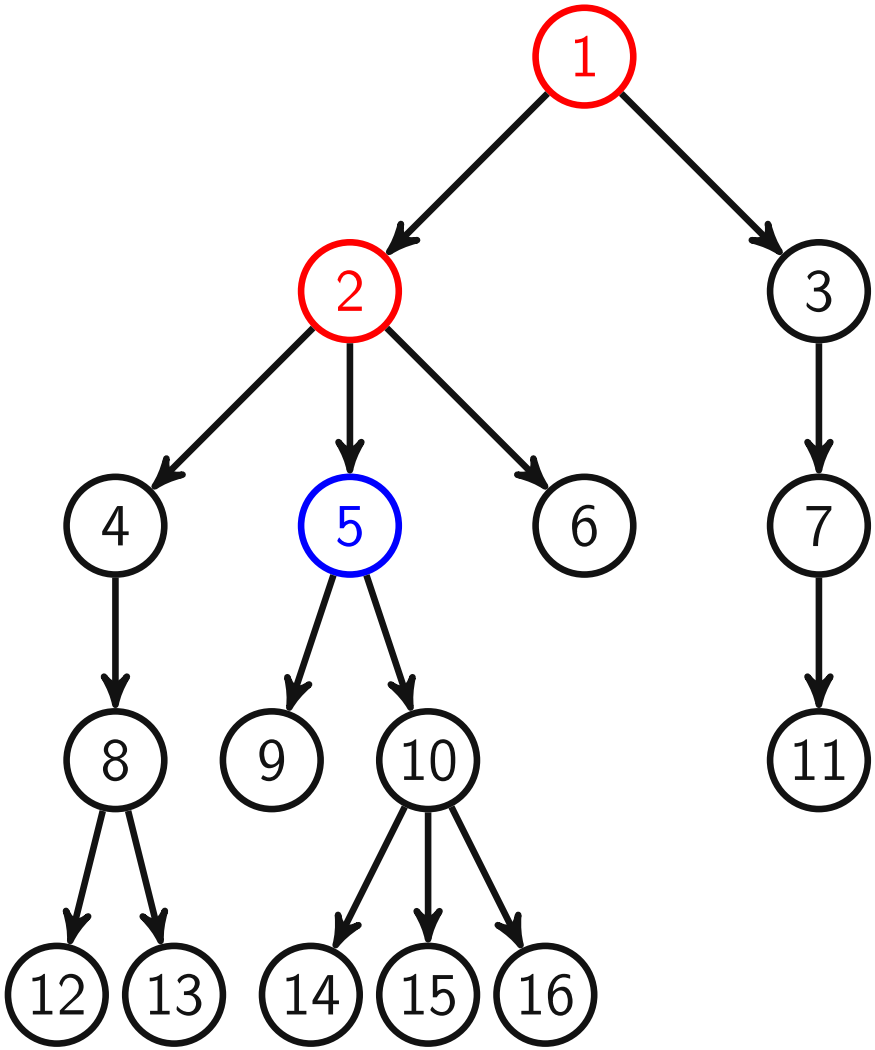
- *Nœud interne* : Un nœud qui possède au moins un fils ;
- *Feuille* : Un nœud sans aucun fils ;
- *Ascendants* : Le père d'un nœud et ses ascendants
- *Descendants* : Les fils d'un nœud et leurs descendants
  (= nœuds du sous-arbre)

Définitions



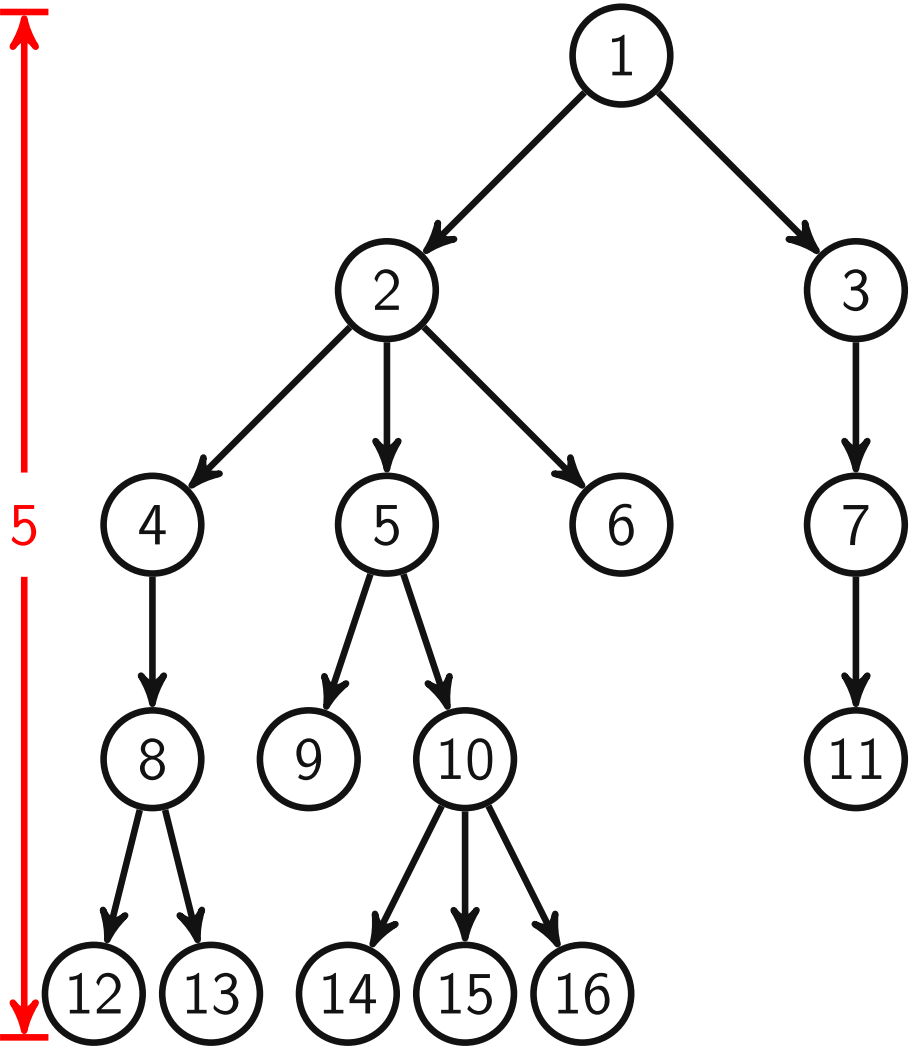
- *Nœud interne* : Un nœud qui possède au moins un fils ;
- *Feuille* : Un nœud sans aucun fils ;
- *Ascendants* : Le père d'un nœud et ses ascendants
- *Descendants* : Les fils d'un nœud et leurs descendants
  (= nœuds du sous-arbre)
- *Chemin* d'un nœud à un descendant

Définitions



- *Nœud interne* : Un nœud qui possède au moins un fils ;
- *Feuille* : Un nœud sans aucun fils ;
- *Ascendants* : Le père d'un nœud et ses ascendants
- *Descendants* : Les fils d'un nœud et leurs descendants
  (= nœuds du sous-arbre)
- *Chemin* d'un nœud à un descendant
- *Profondeur* : nombre d'ascendants d'un nœud

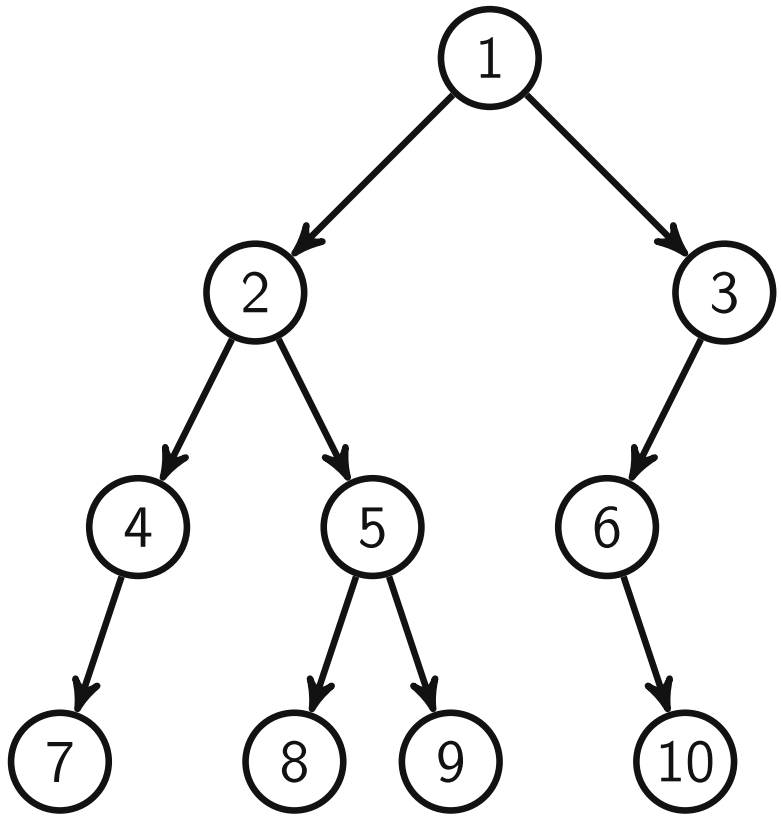
Définitions



- *Nœud interne* : Un nœud qui possède au moins un fils ;
- *Feuille* : Un nœud sans aucun fils ;
- *Ascendants* : Le père d'un nœud et ses ascendants
- *Descendants* : Les fils d'un nœud et leurs descendants
  (= nœuds du sous-arbre)
- *Chemin* d'un nœud à un descendant
- *Profondeur* : nombre d'ascendants d'un nœud
- *Hauteur* de l'arbre vide : 0, d'une feuille : 1, sinon : 1 + hauteur maximale des sous-arbres.

Arbres binaires

- *Arbre binaire* : arbre dont chaque nœud a au plus deux fils : un “fils gauche” et/ou un “fils droit”.



Exercice 1 – Arbres binaires

1. Quelle est la hauteur maximale d'un arbre binaire à n nœuds ?
2. Quel est le nombre maximal de nœuds au niveau k dans un arbre binaire ?
3. Quelle est la hauteur minimale d'un arbre binaire à n nœuds ?
4. Quelle est la particularité du nombre de nœuds internes des arbres binaires *complets* ?
   (Dans un arbre binaire *complet*, tous les nœuds internes ont exactement deux fils)
5. Quel est le nombre de feuilles d'un arbre binaire *complet* à n nœuds ?

Implémentation par pointeurs

Déclaration du type "Arbre binaire"

```
typedef int Element;
typedef struct noeud{
    Element elem;
    struct noeud* filsG;
    struct noeud* filsD;
} Noeud;
typedef Noeud* Arbre;
```

Un arbre binaire est **représenté par un pointeur**

- pour l'arbre vide : on utilise la valeur **NULL**
- sinon le pointeur désigne une **structure** ("noeud") comprenant une étiquette (celle de la racine) et des pointeurs vers ses deux fils (qui peuvent être vides).

Implémentation par pointeurs

Déclaration du type "Arbre binaire"

```
typedef int Element;
typedef struct noeud{
    Element elem;
    struct noeud* filsG;
    struct noeud* filsD;
} Noeud;
typedef Noeud* Arbre;
```

Remarque : pour un arbre non-binaire, il faut pouvoir stocker un nombre plus important de fils...

- soit avec un tableau (si on connait le nombre maximum de fils)
- soit avec une liste chainée

Implémentation par pointeurs

Déclaration du type "Arbre binaire"

```
typedef int Element;
typedef struct noeud{
    Element elem;
    struct noeud* filsG;
    struct noeud* filsD;
} Noeud;
typedef Noeud* Arbre;
```

Remarque : pour un arbre non-binaire, il faut pouvoir stocker un nombre plus important de fils...

- soit avec un tableau (si on connait le nombre maximum de fils)
- soit avec une liste chainée

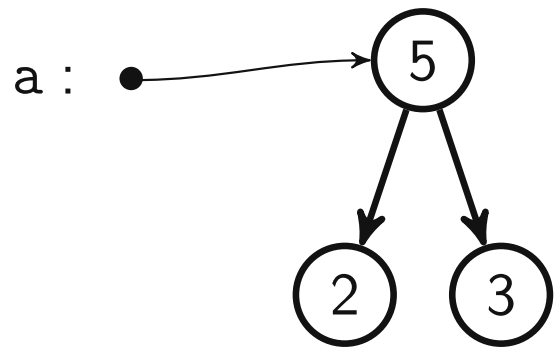
Dans le reste du cours, on ne considère que des arbres binaires.

Fonction de construction d'arbre

```
Arbre newTree(Element elem, Arbre filsG, Arbre filsD){
    Arbre a = malloc(sizeof(Noeud));
    if (a == NULL) return NULL;
    a->elem = elem;
    a->filsG = filsG;
    a->filsD = filsD;
    return a;
}
```

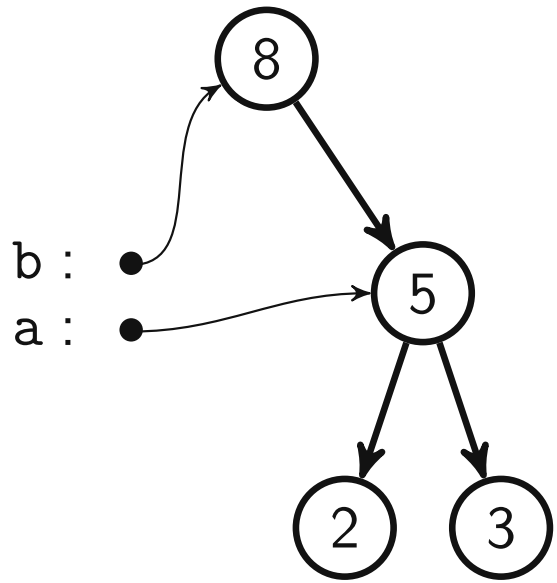
Exemple de construction d'arbre

```
Arbre a = newTree(5,
                  newTree(2, NULL, NULL),
                  newTree(3, NULL, NULL)
                    );

Arbre b = newTree(8, NULL, a );
```



Exemple de construction d'arbre

```
Arbre a = newTree(5,
                  newTree(2, NULL, NULL),
                  newTree(3, NULL, NULL)
                    );

Arbre b = newTree(8, NULL, a );
```

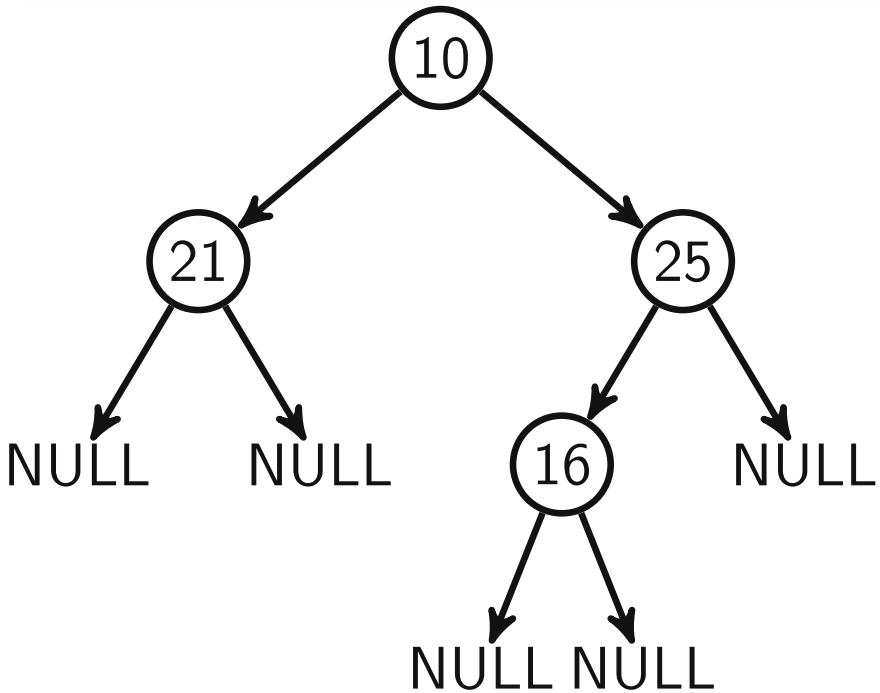


Parcours sur les arbres

- On souhaite “visiter” chaque nœud de l’arbre, pour effectuer une opération (afficher à l’écran, rechercher une étiquette, faire des calculs, etc.)
- L’ordre est important $\Rightarrow$ trois parcours classiques :
  - *Préfixe* : un nœud est visité **avant** ses descendants
  - *Suffixe* : un nœud est visité **après** ses descendants
  - *Infixe* (arbres binaires) : un nœud est visité **entre** son sous-arbre gauche et son sous-arbre droit.
- Dans les trois cas : les nœuds de n’importe quel sous-arbre sont visités consécutivement

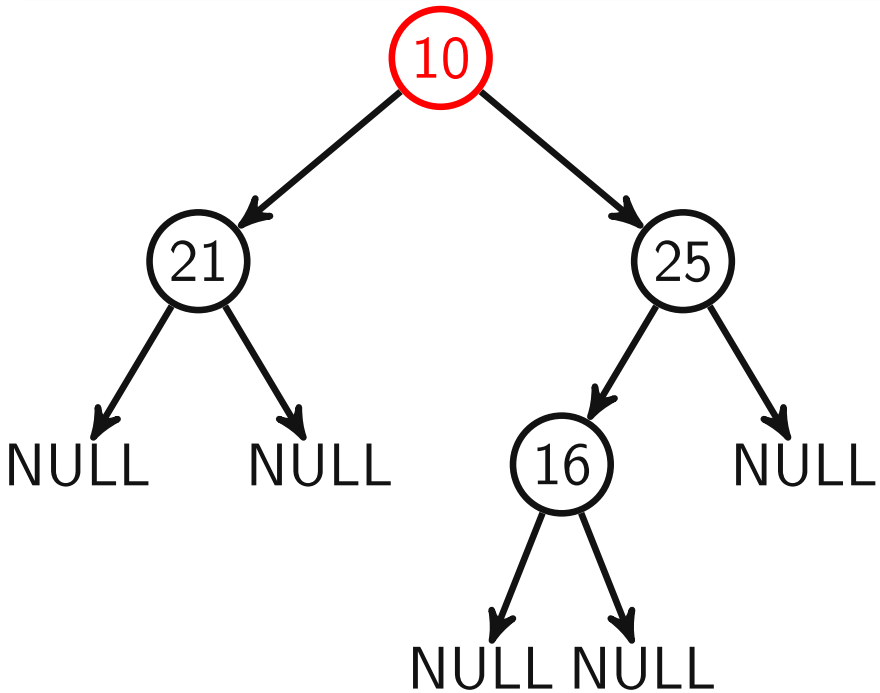
Parcours Préfixe

```
void printPrefix(Arbre a) {
    if (a == NULL) return;
    printf("%d ", a->elem);
    printPrefix(a->filsG);
    printPrefix(a->filsD);
}
```



Parcours Préfixe

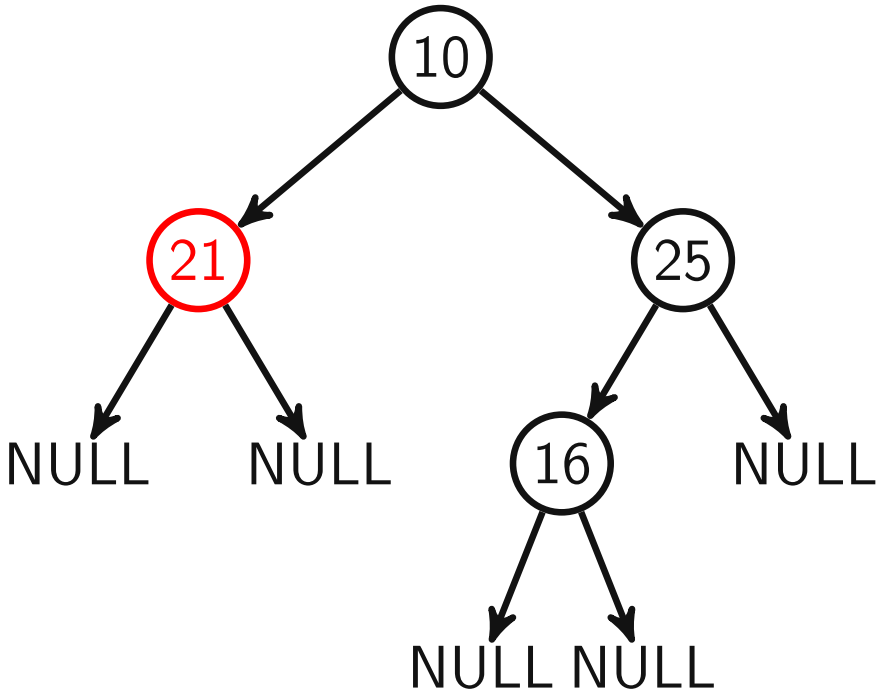
```
void printPrefix(Arbre a) {
    if (a == NULL) return;
    printf("%d ", a->elem);
    printPrefix(a->filsG);
    printPrefix(a->filsD);
}
```




```
>10
```

Parcours Préfixe

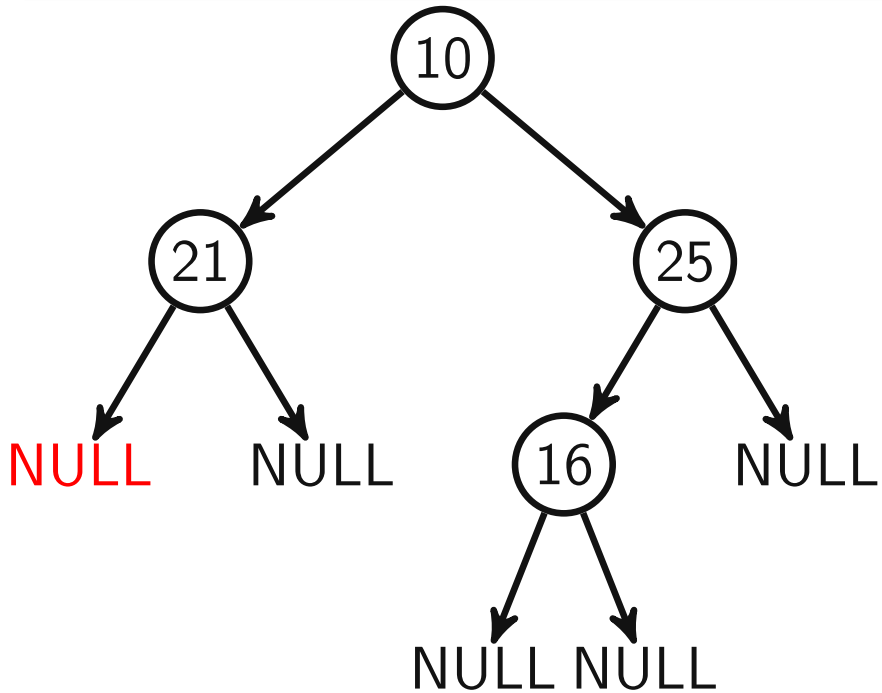
```
void printPrefix(Arbre a) {
    if (a == NULL) return;
    printf("%d ", a->elem);
    printPrefix(a->filsG);
    printPrefix(a->filsD);
}
```




```
>10 21
```

Parcours Préfixe

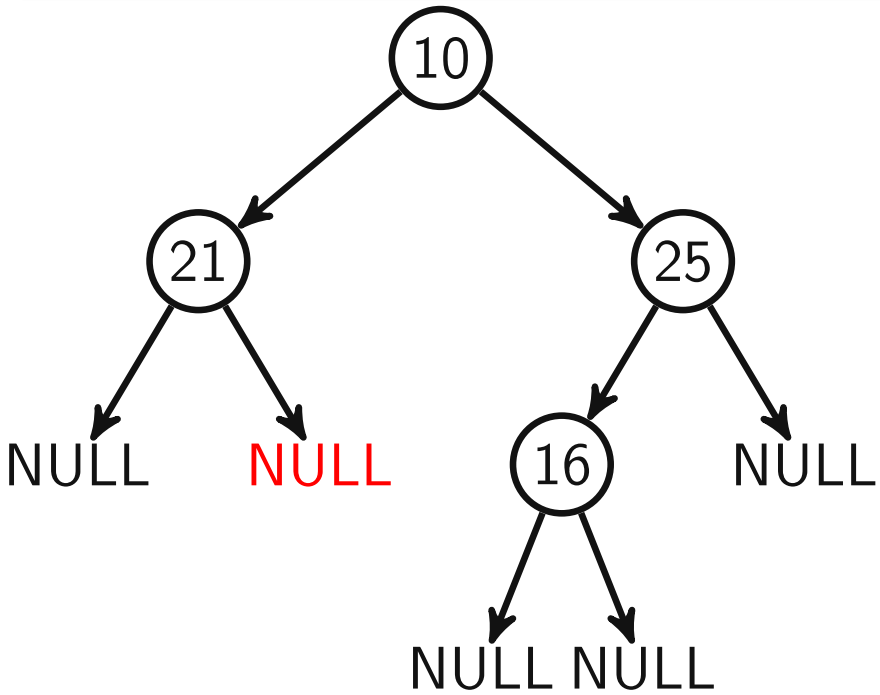
```
void printPrefix(Arbre a) {
    if (a == NULL) return;
    printf("%d ", a->elem);
    printPrefix(a->filsG);
    printPrefix(a->filsD);
}
```




```
>10 21
```

Parcours Préfixe

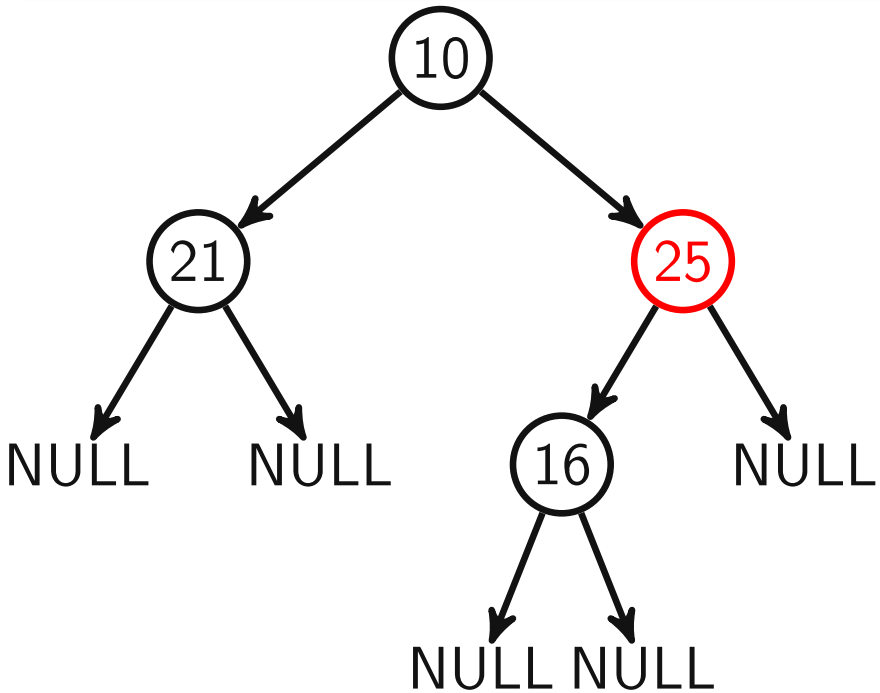
```
void printPrefix(Arbre a) {
    if (a == NULL) return;
    printf("%d ", a->elem);
    printPrefix(a->filsG);
    printPrefix(a->filsD);
}
```




```
>10 21
```

Parcours Préfixe

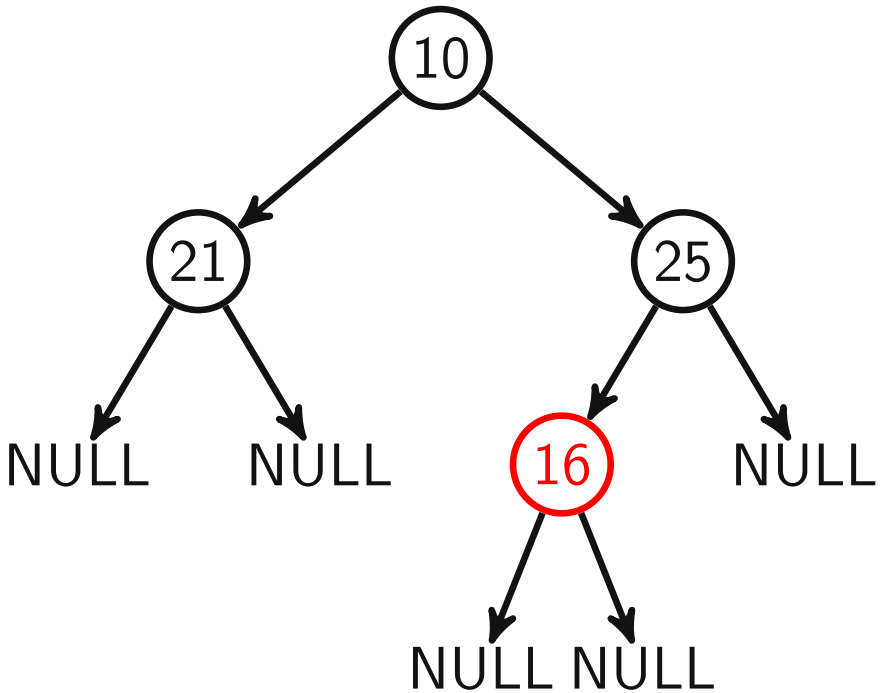
```
void printPrefix(Arbre a) {
    if (a == NULL) return;
    printf("%d ", a->elem);
    printPrefix(a->filsG);
    printPrefix(a->filsD);
}
```




```
>10 21 25
```

Parcours Préfixe

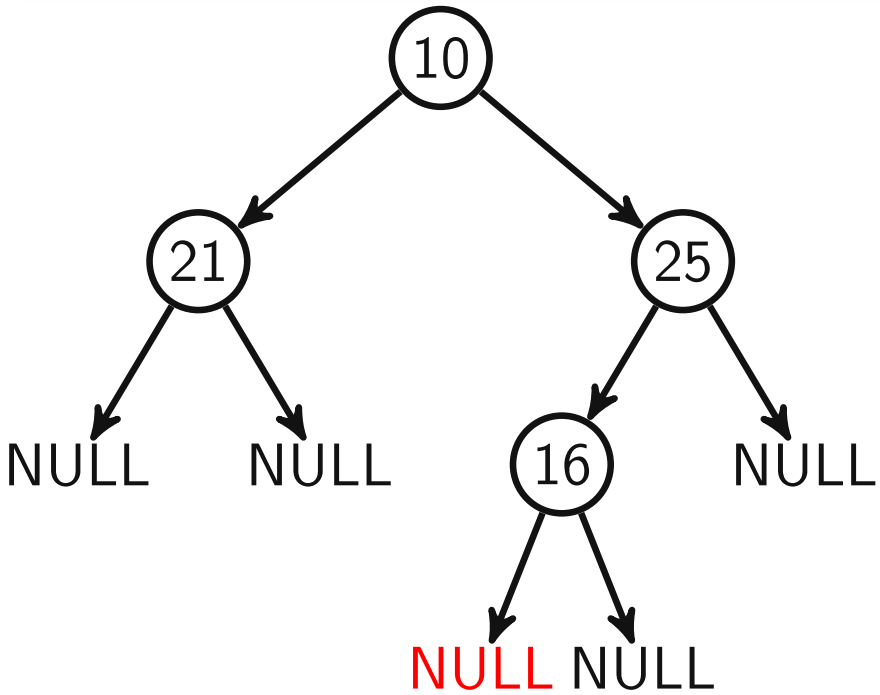
```
void printPrefix(Arbre a) {
    if (a == NULL) return;
    printf("%d ", a->elem);
    printPrefix(a->filsG);
    printPrefix(a->filsD);
}
```




```
>10 21 25 16
```

Parcours Préfixe

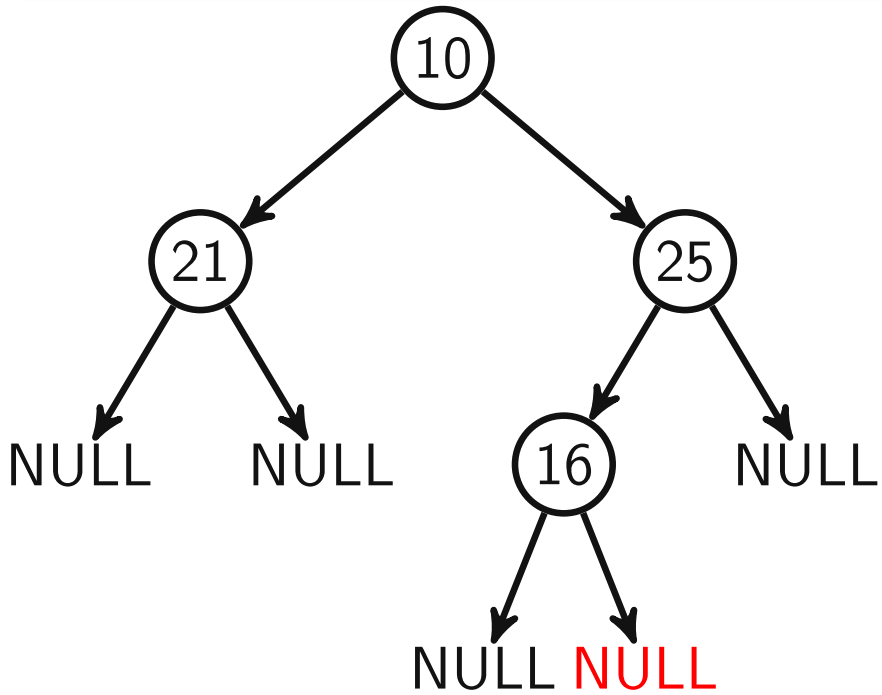
```
void printPrefix(Arbre a) {
    if (a == NULL) return;
    printf("%d ", a->elem);
    printPrefix(a->filsG);
    printPrefix(a->filsD);
}
```




```
>10 21 25 16
```

Parcours Préfixe

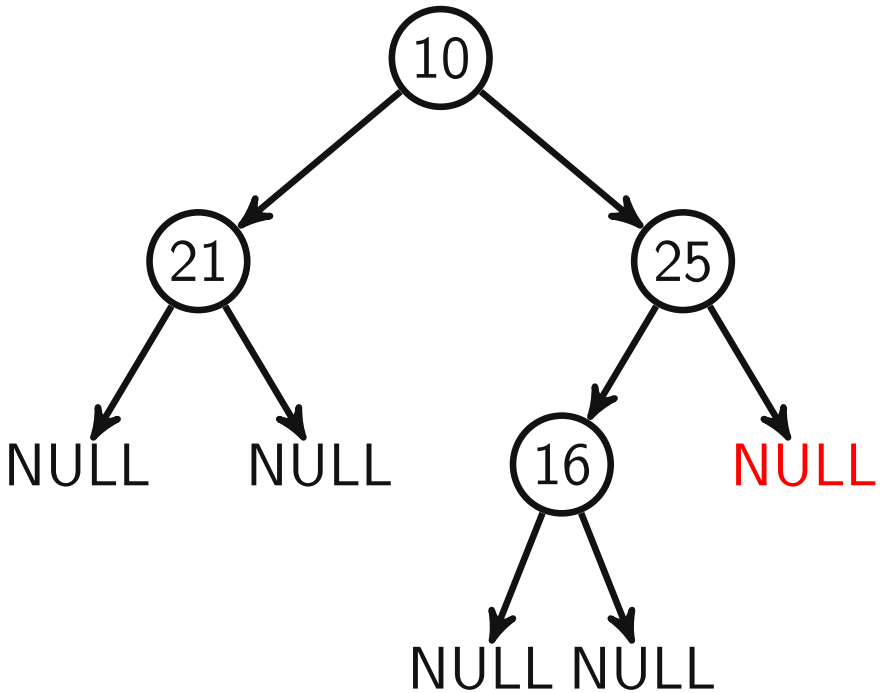
```
void printPrefix(Arbre a) {
    if (a == NULL) return;
    printf("%d ", a->elem);
    printPrefix(a->filsG);
    printPrefix(a->filsD);
}
```




```
>10 21 25 16
```

Parcours Préfixe

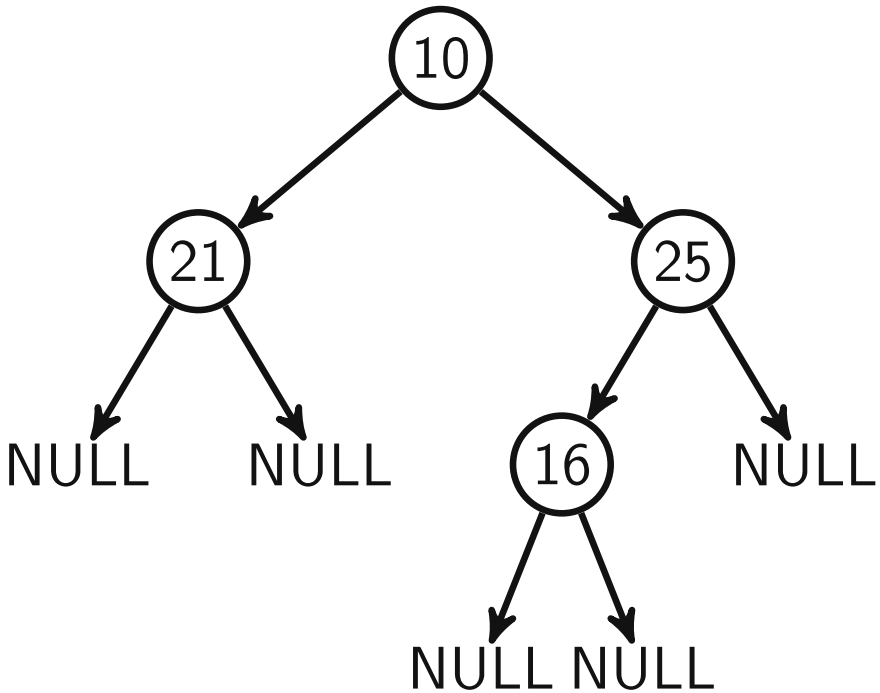
```
void printPrefix(Arbre a) {
    if (a == NULL) return;
    printf("%d ", a->elem);
    printPrefix(a->filsG);
    printPrefix(a->filsD);
}
```




```
>10 21 25 16
```

Parcours Préfixe

```
void printPrefix(Arbre a) {
    if (a == NULL) return;
    printf("%d ", a->elem);
    printPrefix(a->filsG);
    printPrefix(a->filsD);
}
```




```
>10 21 25 16
```

Exercice 2 – Parcours

1. Écrire la fonction `printSuffix(Arbre a)` qui affiche les nœuds de l'arbre en ordre suffixe

2. Écrire la fonction `printInfix(Arbre a)` qui affiche les nœuds de l'arbre en ordre infixe

3. Quelle est la complexité de ces fonctions (en fonction du nombre n de nœuds de l'arbres) ?

Exercice 3 – Fonctions diverses

Écrire les fonctions récursives suivantes. Donner leur complexité en fonction du nombre de nœuds de l'arbre.

1. `int nbLeaves(Arbre a);` Compte le nombre de feuilles de l'arbre binaire `a`.
2. `int height(Arbre a);` Calcule la hauteur de l'arbre binaire `a`.
3. `Arbre copyTree(Arbre a);` Renvoie une copie de l'arbre binaire `a`.
4. `Arbre mirrorTree(Arbre a);` Renvoie une copie miroir de l'arbre binaire `a`.
5. `int levelSize(Arbre a, int prof);`
   Compte le nombre de nœuds de profondeur `prof` de l'arbre binaire `a`.

Exercice 4 – Duos

1. Écrire une fonction qui renvoie le minimum et le maximum des éléments d'un arbre en ne parcourant l'arbre qu'une seule fois.

2. Écrire une fonction qui renvoie le diamètre d'un arbre (le plus long chemin possible, en nombre de nœuds visités, entre deux de ses feuilles).
   Sur l'exemple : 8.