Développement efficace (R3.02)
Récursivité IV : arbres

Marin Bougeret
LIRMM, IUT/Université de Montpellier

Outline

1. Arbre
   - Définition
   - Algorithmes basiques sur les arbres
   - Arbres binaires de recherche

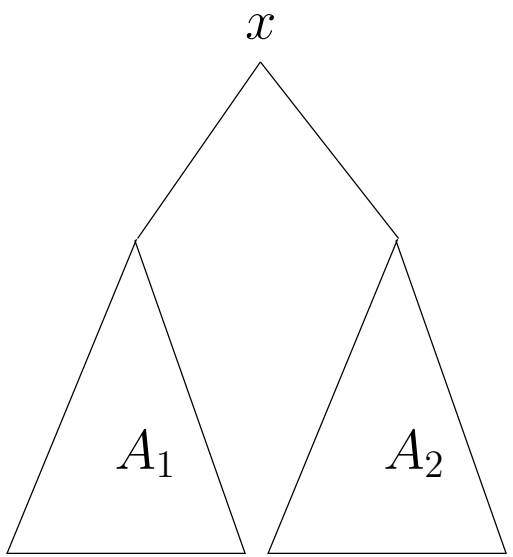
Outline

1. Arbre
   - Définition
   - Algorithmes basiques sur les arbres
   - Arbres binaires de recherche

Définition maths

Un arbre est

- soit vide (noté ())
- soit constitué d'une racine $x \in \mathbb{Z}$, et de 2 sous arbres A_1 et A_2



Définition maths

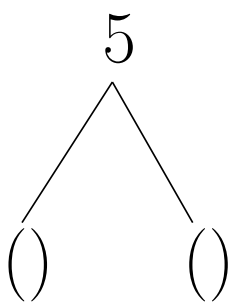
Un arbre est

- soit vide (noté ())
- soit constitué d'une racine $x \in \mathbb{Z}$, et de 2 sous arbres A_1 et A_2

Exemple 1

Comment dessiner l'arbre ayant seulement 5 ?
Plus précisément, cet arbre aura

- 5 pour racine
- un sous arbre gauche vide et un sous arbre droit vide



Définition maths

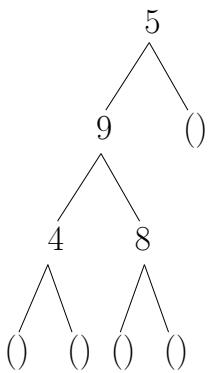
Un arbre est

- soit vide (noté ())
- soit constitué d'une racine $x \in \mathbb{Z}$, et de 2 sous arbres A_1 et A_2

Exemple 2

Comment dessiner l'arbre ayant 5 pour racine, puis

- un sous arbre droit vide
- un sous arbre gauche constitué de 9, ayant lui même 4 à gauche et 8 à droite



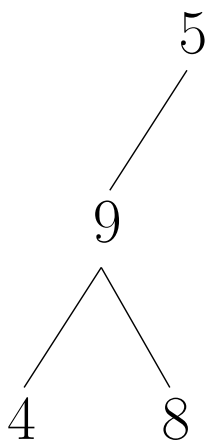
Définition maths

Un arbre est

- soit vide (noté ())
- soit constitué d'une racine $x \in \mathbb{Z}$, et de 2 sous arbres A_1 et A_2

Remarque

- par abus de notation, nous ne dessinerons plus les arbres vides
- le dessin précédent devient ainsi :



Définition en Java

```
class Arbre{
  private int val;
  private Arbre filsG;
  private Arbre filsD;
  //invariant : filsG==null <=> filsD==null

  public Arbre(){//construit l'arbre vide
    this.filsG = null;
    this.filsD = null;
  }
  boolean estVide()
     return (this.filsG==null);
     //vu l'invariant, pas besoin d'ajouter "&&(
        this.filsD==null)";
}
```

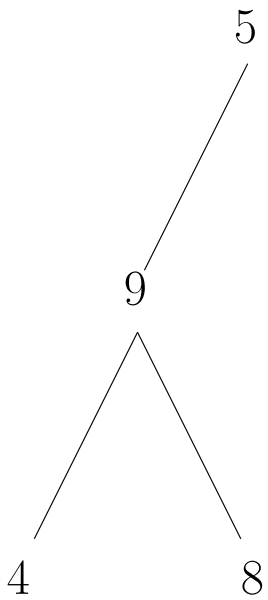
Définition en Java

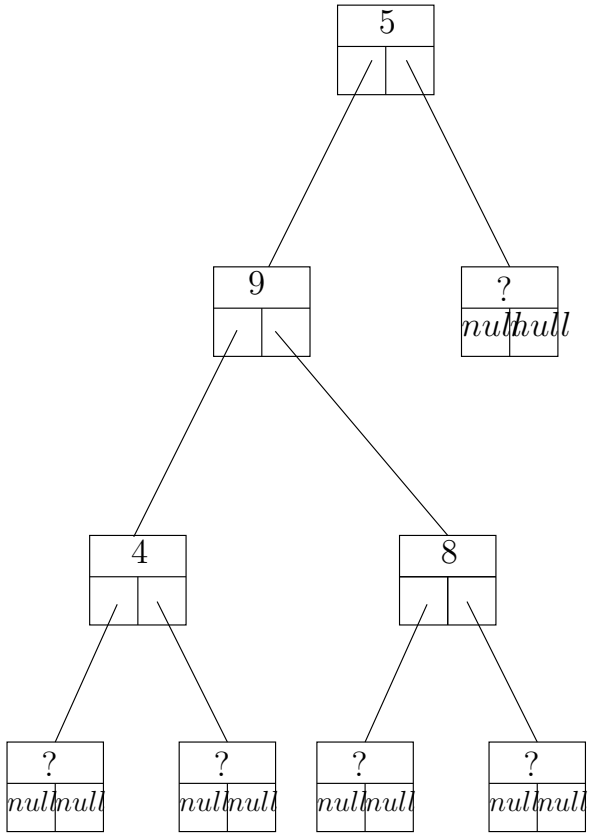
Remarque sur l'arbre vide

D'après le code précédent, notre convention est la suivante :

- l'arbre binaire vide est représentée par n'importe quel objet a de type Arbre avec $a.filsG == a.filsD == null$ (peut importe $a.val$).
- Arbre a = new Arbre() représente donc l'arbre vide
- Arbre a = null ne représente pas l'arbre vide ($a.methode()$ ..)

Exemple






```
Arbre avide = new Arbre();
Arbre a9 = ...;
Arbre a5 = new Arbre();
a5.val=5;
a5.filsG=a9;
a5.filsD=avide;//et pas null
```

Outline

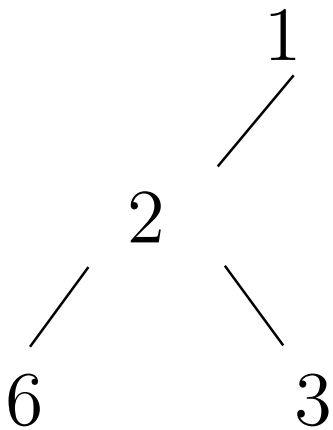
1. Arbre
   - Définition
   - Algorithmes basiques sur les arbres
   - Arbres binaires de recherche

Nombre de sommets (= d'entiers) dans l'arbre

```
int nbSommets(){
  if(estVide()){
    return 0;
  }
  else{
   return 1+filsG.nbSommets()+filsD.nbSommets();
  }
}
```

Hauteur de l'arbre

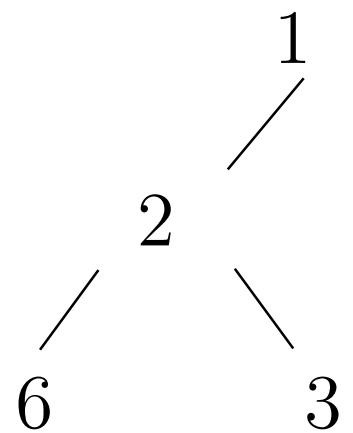
```
int hauteur(){
 if(estVide()){
   return 0;
 }
 else{
 return
 }
}
```



arbre de hauteur 3

Hauteur de l'arbre

```
int hauteur(){
 if(estVide()){
   return 0;
 }
 else{
 return 1+max(filsG.hauteur(),filsD.hauteur());
 }
}
```



arbre de hauteur 3

Recherche dans un arbre

```
boolean recherche(int x){
if(estVide()){
 return false;
}
else{
return filsG.recherche(x) || filsD.recherche(x);
}
}
```

Remarque de complexité

Nous avions vu que, par exemple :

- faire 2 appels récursifs sur $(n-1)$ éléments chacun conduit à une complexité en au moins 2^n, et donc à des algorithmes quasi-inutilisables
- faire 2 appels récursifs sur $(n/2)$ éléments chacun conduit à une complexité raisonable (par exemple, $\mathcal{O}(n), \mathcal{O}(nlog(n))$), et à des algorithmes utilisables

Dans l'exemple précédent :

- `recherche` fait 2 appels récursifs .. est-il inutilisable ? quelle est sa complexité ?

Remarque de complexité

Propriété

La complexité de `recherche` est $\mathcal{O}(n)$, où n est le nombre de sommets de l'arbre (c'est donc tout à fait raisonable)!

Pourquoi $\mathcal{O}(n)$?

Trois façons de se convaincre :

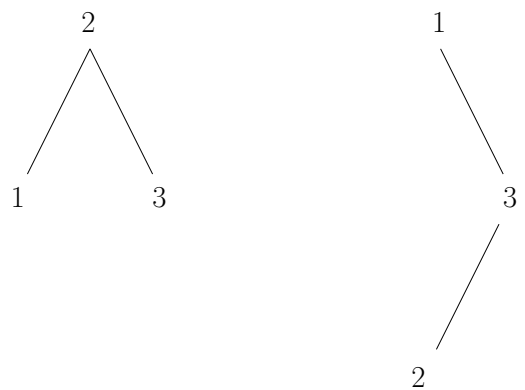
- penser à ce qui se passe à l'éxecution : on éxecute bien un nombre constant d'opérations sur chaque noeud de l'arbre.
- il n'y a qu'un seul appel récursif lancé sur chaque sommet
- on fait un appel récursif à gauche sur n_G éléments, un à droite sur n_D sommets, avec $n_G + n_D = n - 1$ (donc "ressemble" au divide and conquer quand $n_G = n_D \approx \frac{n}{2}$)

toString()

```
String toStringNaif(){
 if(estVide())
  return "()";
 else
   return filsG.toStringNaif()+"␣"+this.val+"␣"+
       filsD.toStringNaif();
}
```

Problème

- plusieurs arbres différents peuvent donner la même chaîne!
- ex : la chaîne "() 1 () 2 () 3 ()" peut provenir des deux arbres suivants :

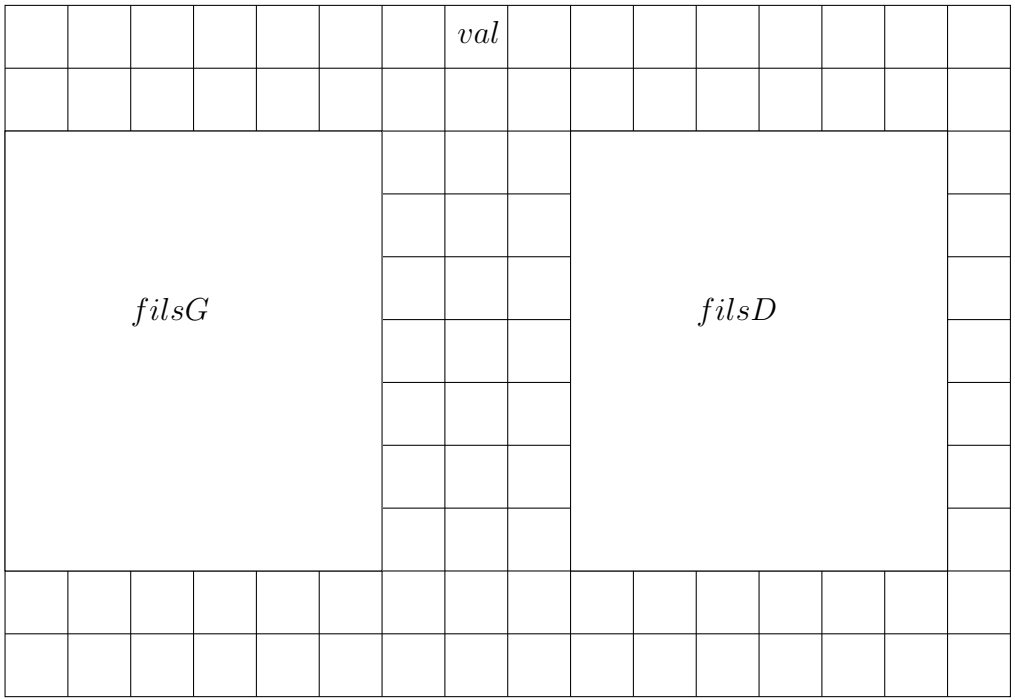


toString()

Une solution

On va donc "dessiner" l'arbre dans le terminal

- pb : on ne peut pas le dessiner facilement récursivement avec la racine en haut
- solution : on tourne la tête de 90^o vers la gauche!

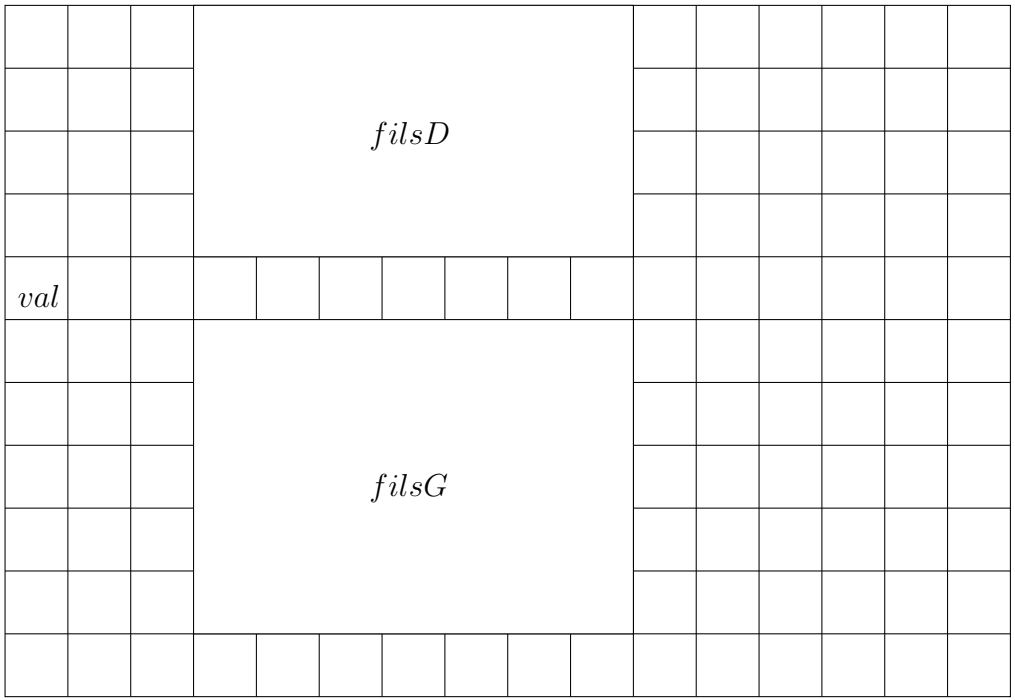


toString()

Une solution

On va donc "dessiner" l'arbre dans le terminal

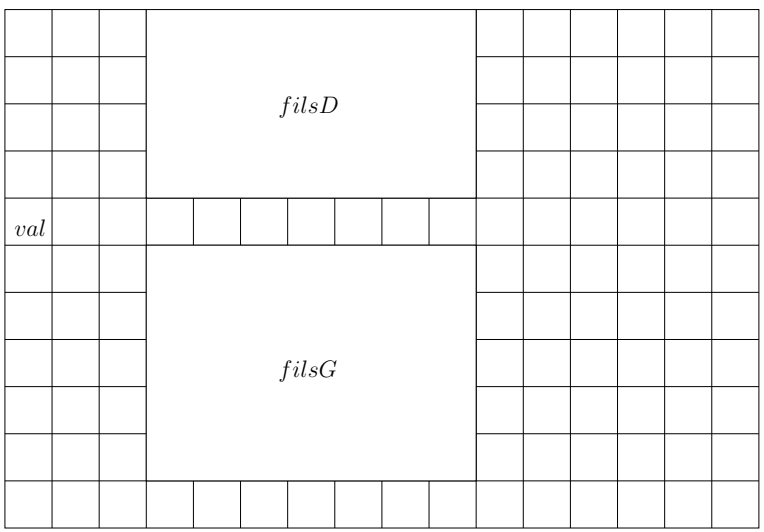
- pb : on ne peut pas le dessiner facilement récursivement avec la racine en haut
- solution : on tourne la tête de 90° vers la gauche!



toString()

Deux précisions importantes

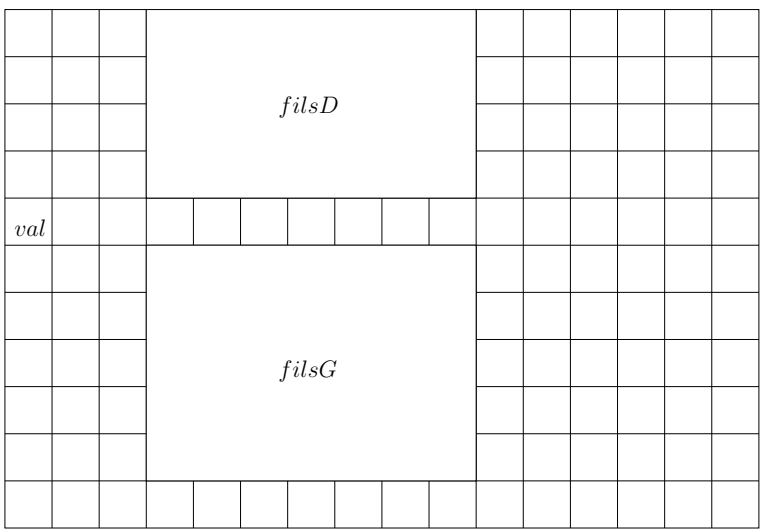
- il ne faut pas simplement demander récursivement "affiche toi", mais "affiche toi avec 3 espaces à chaque début de ligne"
  - on prend donc en paramètre une chaîne s contenant les espaces à ajouter au début de chaque ligne
  - la phrase précédente devient "affiche toi avec 3 espaces de plus dans s à chaque début de ligne"



toString()

Deux précisions importantes

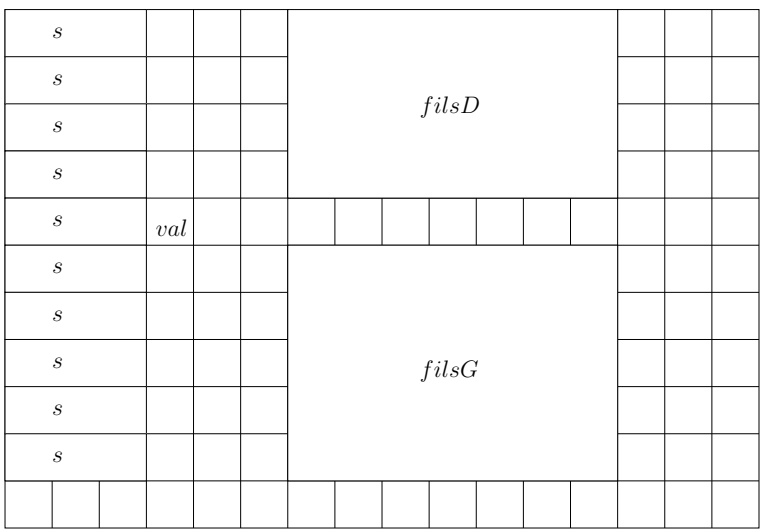
- il ne faut pas simplement demander récursivement "affiche toi", mais "affiche toi avec 3 espaces à chaque début de ligne"
  - on prend donc en paramètre une chaîne s contenant les espaces à ajouter au début de chaque ligne
  - la phrase précédente devient "affiche toi avec 3 espaces de plus dans s à chaque début de ligne"



toString()

Deux précisions importantes

- il ne faut pas simplement demander récursivement "affiche toi", mais "affiche toi avec 3 espaces à chaque début de ligne"
  - on prend donc en paramètre une chaîne s contenant les espaces à ajouter au début de chaque ligne
  - la phrase précédente devient "affiche toi avec 3 espaces de plus dans s à chaque début de ligne"



toString()

```
public String toStringV2aux(String s){
  //pre : aucun
  //resultat : retourne une chaine de caracteres
      permettant d'afficher this dans un
     terminal (avec l'indentation du dessin
     precedent, et en ajoutant s au debut de
     chaque ligne ecrite) et passe a la ligne

  if( estVide ())
    return s+"()\n";
  else
   return filsD.toStringV2aux (s + "␣␣␣") + s +
      val + "\n"+ filsG.toStringV2aux (s + "␣␣␣"
      );
```

toString()

Conclusion

Le toString recherché est donc le suivant.

```
public String toStringV2(){
  //pre : aucun

  return toStringV2aux("");
}
```

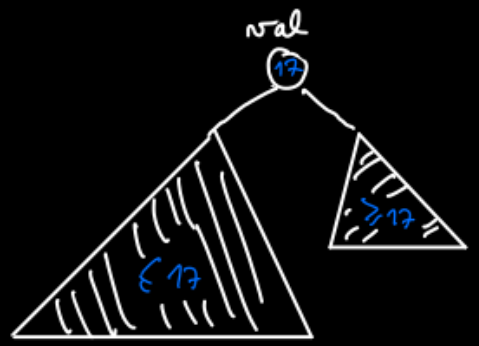
Outline

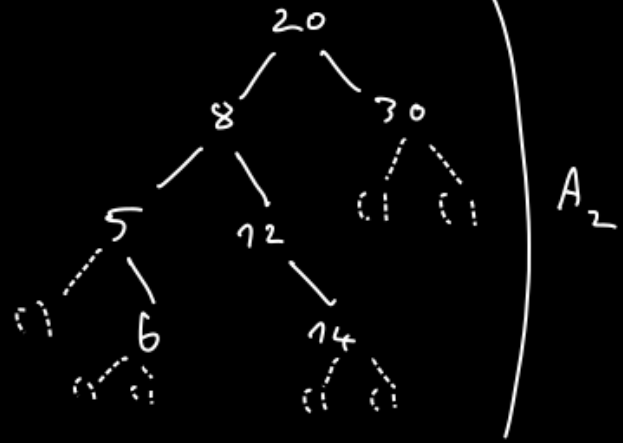
1. Arbre
   - Définition
   - Algorithmes basiques sur les arbres
   - Arbres binaires de recherche

Déf Un ABR (Arbre Binaire de Recherche) est soit vide,
soit constitué de deux ABR A_G, A_D et d'une racine $val \in \mathbb{Z}$ /

pour toute valeur $x \in A_G$: $x \leq val$

pour toute valeur $x \in A_D$: $val \leq x$





A_1 n'est pas un ABR

(car A_G n'est pas un ABR)

A_2 est un ABR

En Java : la classe ABR est comme la classe Arbre !

```
class ABR {
    private int val;
    private ABR filsG;
    private ABR filsD;

    public ABR () {
        filsG = null;
        filsD = null;
    }

    ...

}
```

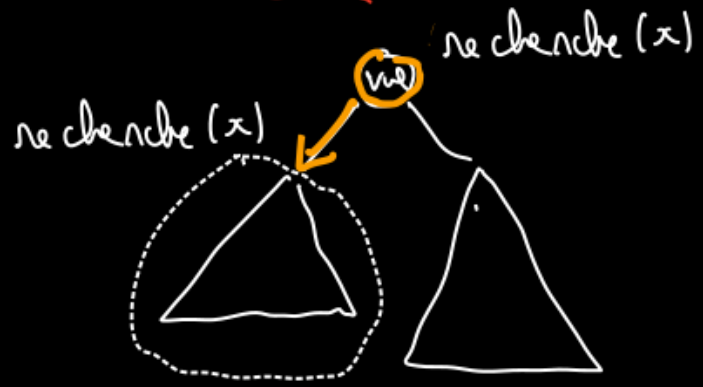
Intérêt des ABR : la recherche est rapide.

Stratégie pour `boolean recherche(int x)` :

Si `x == val`, c'est fini

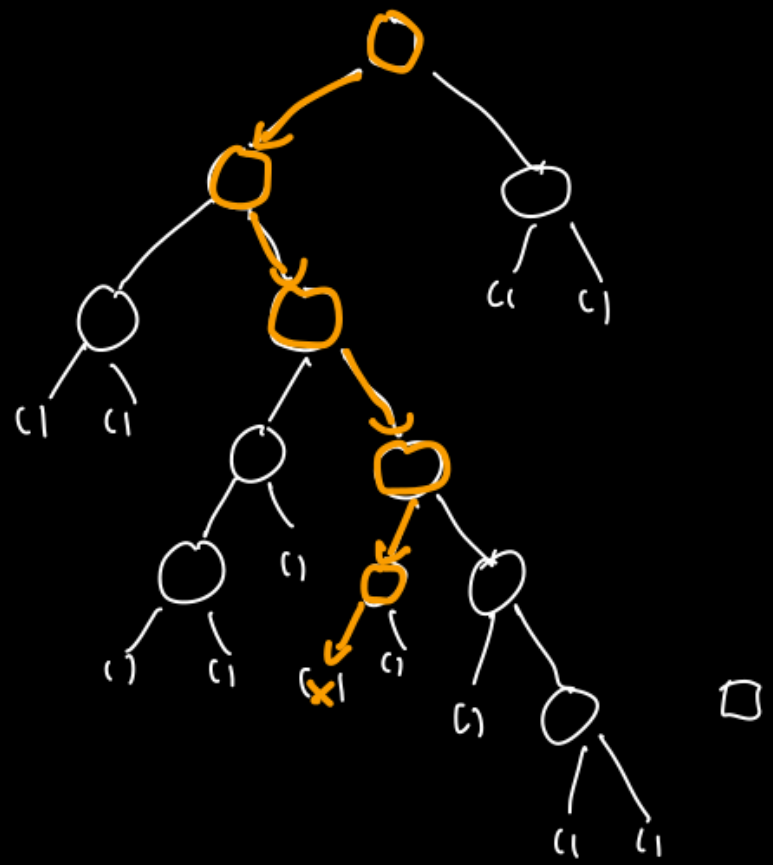
Si `x < val`, il suffit de rechercher récursivement à gauche seulement

Si `x > val`, pareil, à droite seulement



Théorème recherche s'effectue en $\boxed{O(h)}$, où h est la hauteur de l'arbre

- "preuve"



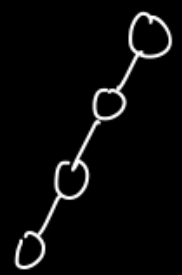
Vous verrez (cf TD) que les autres opérations (insertion, suppression) se font également en $O(h)$.

⇒ Que peut-on dire de h par rapport à n (nb de sommets) ?

Propriété Pour tout arbre à n sommets et de hauteur h, on a : $\boxed{h \leqslant n \leqslant 2^h - 1}$

Cas extrême 1 $h = n$:

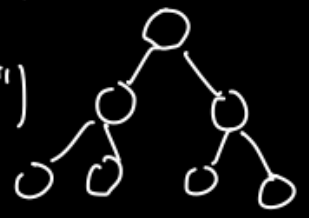
(arbre "chemin")



Cas extrême 2 : $n = 2^{h-1}$

(arbre "complet")

($h = 3$)



$1 + 2 + \dots + 2^{h-1} = 2^h - 1$

□ Preuve de $n \leqslant 2^h - 1$:

1) $n \leqslant m_h$ avec m_h : nombre de sommets d'un arbre complet de hauteur h.

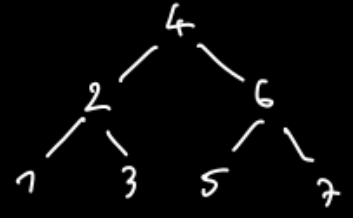
2) $m_h = 2^h - 1$ □

(on avait) $h \leq n \leq 2^h - 1$

Propriété (ré-écrite) : $\lg(n) \leq h \leq n$

Conclusion : quand h est proche de $\lg(n)$ (arbre "équilibré"), les ABR sont très efficaces.

"Mauvais" ABR déséquilibré ($h = n$)



"Bon" ABR équilibré ($h \approx \lg(n)$)

Pour nos ABR, on essaiera pas d'obtenir des arbres équilibrés.

Mais il existe des structures de données permettant d'avoir des ABR équilibrés !