

DIU EIL 2019 / Bloc 5

Algorithmique avancée

Séance 2 – Algorithmique des arbres

Au programme :

- Algorithmes sur les arbres binaires
 - Calculer le nombre de nœuds, la hauteur, ...
 - Rechercher une étiquette
- Arbres binaires de recherche
 - Principe d'organisation, propriétés
 - Algorithme de recherche
 - Algorithme d'insertion
 - Autres algorithmes

Considérations générales

- Comme indiqué à la séance précédente, la majorité des algorithmes sur les arbres s'écrivent naturellement de façon récursive : un arbre binaire a deux sous-arbres binaires pour fils, la structure est naturellement inductive.
- La plupart des algorithmes reviennent à réaliser un parcours de l'arbre (cf. préfixe, infixe, suffixe) avec un traitement appliqué en chaque nœud. Les questions à se poser sont :
 - Quel ordre de traitement entre le nœud courant et ses fils ?
E.g., faut-il connaître un résultat provenant des fils ?
 - Faut-il explorer systématiquement les fils ? Autrement dit, le parcours de l'arbre peut-il être partiel ?
 - Que faire au niveau des feuilles ? Ou, plus atomique encore, au niveau d'un arbre vide ?

Séance 2 – Algorithmique des arbres

2.1 – Algorithmes de calcul

Calcul du nombre de nœuds (1)

- Ici, il s'agit simplement de compter les nœuds rencontrés lors du parcours
 - Le parcours doit donc être exhaustif
 - Chaque nœud compte pour 1, même les feuilles
 - Le nombre de nœud dans un arbre non-vide vaut :
 $1 \text{ (racine)} + \text{nb nœuds dans ses sous-arbres gauche et droit}$

Calcul du nombre de nœuds (2)

- Algorithme (v1) :

```
fonction nbNœuds(ArbreBinaire A) → Entier
  variables Entiers ng, nd
  ng ← 0
  si non estVide(gauche(A)) alors
    ng ← nbNœuds(gauche(A))
  finsi
  nd ← 0
  si non estVide(droit(A)) alors
    nd ← nbNœuds(droit(A))
  finsi
  retourner 1 + ng + nd
```

- Cet algorithme a une précondition : l'arbre A ne doit pas être vide, autrement les préconditions des opérations gauche et droit sont violées !

Calcul du nombre de nœuds (3)

- Un arbre vide n'a pas de nœuds \Rightarrow il compte pour 0
- Algorithme (v2) :

```
fonction nbNœuds (ArbreBinaire A)  $\rightarrow$  Entier  
  si estVide(A) alors  
    retourner 0  
  sinon  
    retourner 1 + nbNœuds (gauche(A))  
                + nbNœuds (droit(A))  
  finsi
```

- Questions :
 - Cet algorithme est-il correct ?
 - Quelle est sa complexité ?

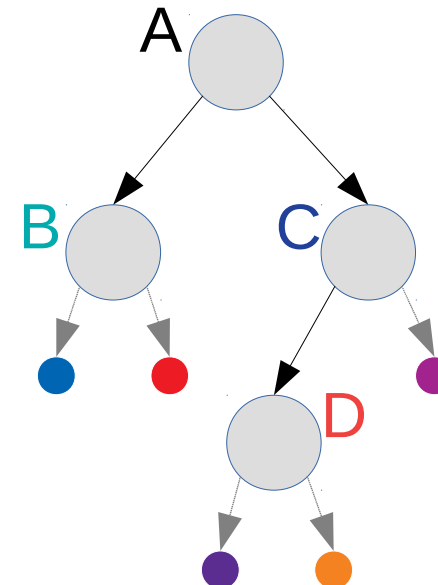
Calcul du nombre de nœuds (4)

- Approche « intuitive » :
 - Correct car ...
 - il termine (les sous-arbres gauche et droit sont strictement « plus petits »
⇒ la récursion fini par atteindre un arbre vide ;
 - et il calcule correctement le nombre de nœuds, cf. relation de récurrence posée initialement.
 - Temps en $O(N)$ pour un arbre à N nœuds car ...
 - le temps de calcul de chaque appel est constant, i.e., $O(1)$;
 - et chaque nœud de l'arbre correspond à un appel ...
+ 1 appel pour chaque « fils manquant » (arbre vide gauche et/ou droit).
Or, le nombre de « fils manquants » vaut $N+1$ dans un arbre binaire à N nœuds (*à démontrer !*)
- Une démonstration plus formelle se ferait par induction structurelle ... ou plus simplement par récurrence. (*hors programme !*)

Calcul du nombre de nœuds (5)

- Une propriété intéressante : les appels récurifs suivent la structure de l'arbre !
- Exemple :

- Appel initial : nbNœuds(A)
 - Appelle nbNœuds(B)
 - Appelle nbNœuds(vide) // gauche
→ retourne 0
 - Appelle nbNœuds(vide) // droit
→ retourne 0
 - retourne 1+0+0
 - Appelle nbNœuds(C)
 - Appelle nbNœuds(D)
 - Appelle nbNœuds(vide) // gauche
→ retourne 0
 - Appelle nbNœuds(vide) // gauche
→ retourne 0
 - retourne 1+0+0
 - Appelle nbNœuds(vide) // droit
→ retourne 0
 - retourne 1+1+0
- retourne 1+1+2



Exercice (30')

- En suivant les principes vus pour le calcul du nombre de nœuds, proposer un algorithme récursif qui détermine la **hauteur** d'un arbre binaire
- Le simuler sur un exemple
- Bien préciser sa précondition
- Argumenter sa correction et sa complexité

Éléments de solution : calculer la hauteur (1)

- Principe
 - Traitement en chaque nœud :
 - Calculer la hauteur h_g du sous-arbre gauche
 - Calculer la hauteur h_d du sous-arbre droit
 - La hauteur de l'arbre vaut $1 + \max(h_g, h_d)$
 - Traitement terminal :
 - La hauteur d'un arbre à un seul nœud est 0 ... mais cela obligerait à tester la présence des fils (cf. calcul du nombre de nœuds)
 - Cependant la hauteur d'un arbre vide est mal définie !
→ on la considérera comme valant -1 par cohérence avec la formule de récurrence
Ainsi, la hauteur d'une feuille sera $1 + \max(-1, -1) = 0$

Éléments de solution : calculer la hauteur (2)

- Algorithme :

```
fonction hauteur(ArbreBinaire A) → Entier
  si estVide(A) alors
    retourner -1
  sinon
    retourner 1 + max( hauteur(gauche(A)),
                      hauteur(droit(A)) )
  finsi
```

- Propriétés :

- Précondition : l'arbre initial ne doit pas être vide (hauteur mal définie dans ce cas)
- Correction : cf. argumentaire nbNoeuds
- Complexité : idem

Algorithmes de calcul : conclusion

- Les algorithmes de calcul nécessitent généralement un parcours exhaustif de l'arbre
⇒ leur complexité temporelle est au moins en $O(N)$
- La forme de l'algorithme est souvent très similaire et reprend celle d'un parcours suffixe (il faut avoir fait le calcul sur les sous-arbres pour déterminer le résultat)

Séance 2 – Algorithmique des arbres

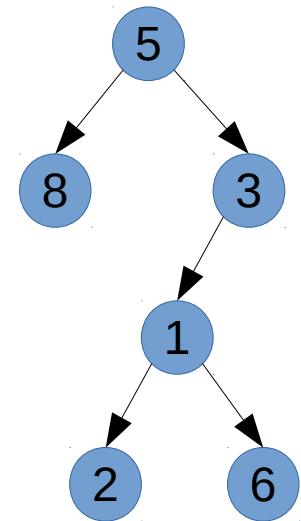
2.2 – Algorithmes de recherche

Recherche dans un arbre binaire (1)

- On veut vérifier qu'une étiquette E donnée est présente dans un nœud de l'arbre \Rightarrow il faut le parcourir.
- Algorithme (v1) :

```
- fonction etqPrésente(ArbreBinaire A, Etiquette E)  $\rightarrow$  Booléen  
  variables Booléens ePg, ePd  
  si estVide(A) alors  
    retourner Faux  
  sinon  
    ePg  $\leftarrow$  etqPrésente(gauche(A), E)  
    ePd  $\leftarrow$  etqPrésente(droit(A), E)  
    retourner racine(A)=E ou ePg ou ePd  
  finsi
```

- Tester sur l'arbre ci-contre :
 etqPrésente(A,2) ; etqPrésente(A,4) ; etqPrésente(A,5)
 Que constatez vous ?

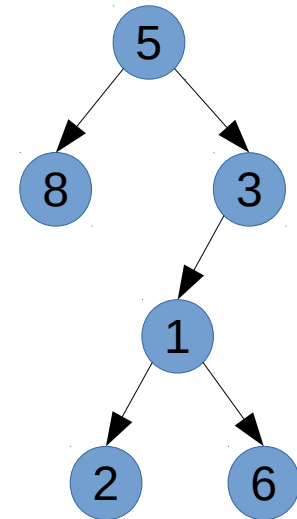


Recherche dans un arbre binaire (2)

- On peut faire mieux : une fois l'étiquette trouvée, inutile de poursuivre !
- Algorithme (v2) :

```
fonction etqPrésente(ArbreBinaire A, Etiquette E) → Booléen  
  si estVide(A) alors  
    retourner Faux  
  sinon  
    retourner racine(A)=E  
      ou etqPrésente(gauche(A), E)  
      ou etqPrésente(droit(A), E)  
  finsi
```

- Remarques :
 - Il faut considérer des « *ou* » à *évaluation paresseuse* (ou bien décomposer en conditionnelles imbriquées)
 - Appeler sur gauche ou droit en premier ne change rien
- Tester à nouveau sur l'arbre ci-contre :
etqPrésente(A,2) ; etqPrésente(A,4) ; etqPrésente(A,5)
Que constatez vous ?



Recherche dans un arbre binaire (3)

- Propriétés de la v2 :
 - Correction et terminaison : cf. argumentation nbNœuds (c'est toujours un parcours)
 - Complexité : temps pour un arbre à N nœuds en ... $O(N)$ **?!?!**

On n'a rien gagné en **pire cas** :

- Si l'étiquette est absente, il faut bien regarder tous les nœuds pour conclure

Mais on amélioré le **meilleur cas** :

- Si l'étiquette est dans la racine de l'arbre initial, l'algorithme se termine en temps constant.
- Il s'agit d'un **minorant** de la complexité de cet algorithme, on le note avec le **symbole omega** : $\Omega(1)$
- En améliorant le meilleur cas d'un algorithme, on améliore généralement ses performances moyennes

Exercice : des problèmes (pas si) voisins (30')

- Discuter les problèmes de recherche suivants
 - Trouver l'étiquette la plus grande dans l'arbre (éventuellement, en tenant compte d'une borne inférieure imposée, cf. bloc 2)
 - Compter le nombre d'occurrences d'une étiquette donnée
- En particulier :
 - S'agit-il de problème traitables par un parcours partiel ?
 - Quels sont les algorithmes ?
 - Quelles sont leurs complexités ?

Recherche dans un arbre binaire : conclusion

- Dans une liste ou dans un arbre, la recherche prend donc un temps similaire
- Mais dans une liste, sous hypothèse de tri, on pouvait faire mieux : une recherche dichotomique !
- Peut-on faire de même dans un arbre binaire ? Autrement dit, quelle hypothèse d'ordre faire pour permettre une recherche plus efficace ?

Séance 2 – Algorithmique des arbres

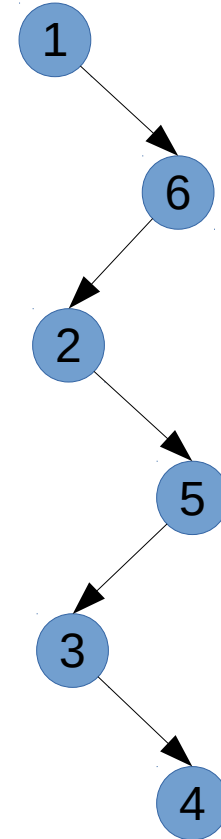
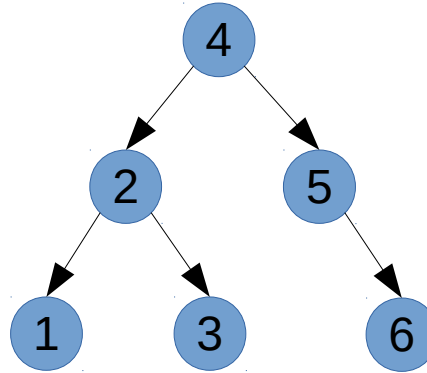
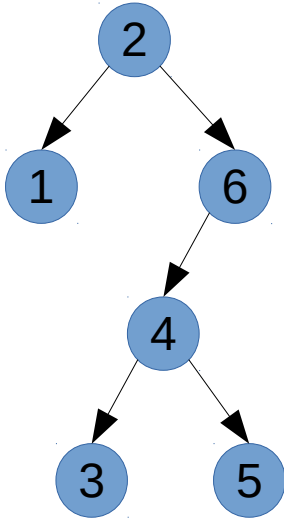
2.3 – Arbres de recherche

Arbre binaire de recherche

- Un **arbre de recherche** est un arbre dont les étiquettes sont organisées de façon à respecter une **relation d'ordre partiel**
- Dans le cas d'un arbre binaire de recherche (ABR), tout nœud a une étiquette
 - plus grande ou égale à celles dans son sous-arbre gauche
 - plus petite strictement que celles dans son sous-arbre droit

Exemples

- Voici quelques exemples d'ABR :



- Exercice :
 - Construisez 3 arbres de forme quelconque à 10 nœuds ; rangez-y les entiers de 1 à 10 à la façon d'un ABR. Que constatez-vous ?

ABR : un premier algorithme

- On veut vérifier qu'un arbre binaire est un ABR, mais attention : vérifier la propriété *locale* en chaque nœud ne suffit pas !
- Principe :
 - La propriété est vérifiée en un nœud ssi
 - soit il n'a pas de fils gauche (resp. droit) ;
 - soit la plus grande (resp. petite) étiquette de celui-ci est inférieure ou égale (resp. supérieure) à la sienne.
⇒ il faut disposer des valeurs min/max à gauche/droite.
 - Une arbre binaire vide est-il un ABR ? Réponse : oui !
Quel est son min/max ? Réponse : $+\infty/-\infty$

ABR : un premier algorithme

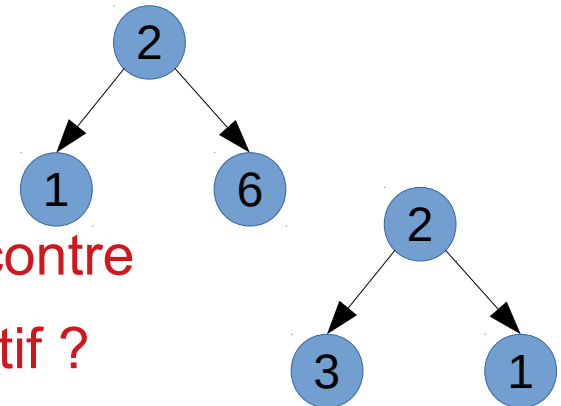
- Algorithme :

```
fonction estABR(ArbreBinaire A) → (Booléen, Entier, Entier)
  variables Booléens abrG, abrD ; Entiers minG, maxG,
                                     minD, maxD

  si estvide(A) alors
    retourner (Vrai,  $+\infty$ ,  $-\infty$ )
  sinon
    (abrG, minG, maxG) ← estABR(gauche(A))
    (abrD, minD, maxD) ← estABR(droite(A))
    retourner (racine(A) ≥ maxG et racine(A) < minD),
               min(minG, racine(A)), max(maxD, racine(A))
  finsi
```

- Questions :

- Appliquer cet algorithme aux arbres ci-contre
- S'agit-il d'un parcours partiel ou exhaustif ?



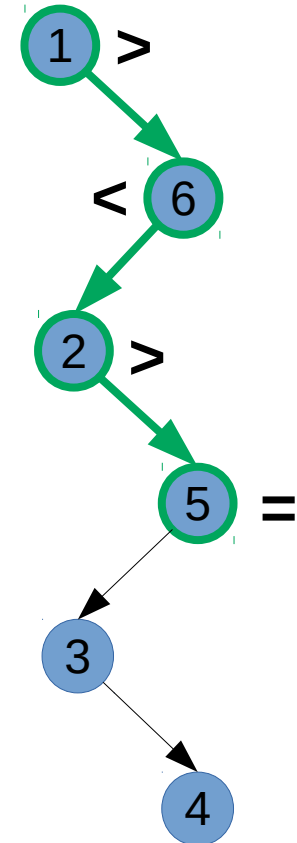
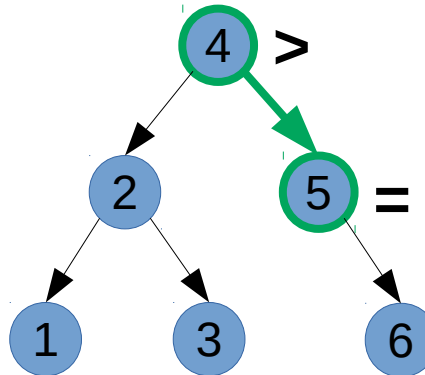
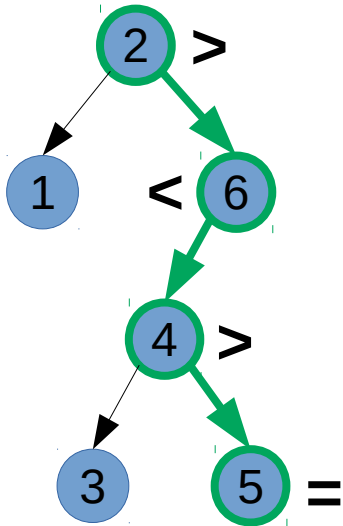
Recherche dans un ABR (1)

- La propriété d'ordre en chaque nœud d'un ABR assure qu'il existe un unique chemin pour toute valeur stockée : la comparaison en chaque nœud indique si la recherche doit être poursuivie à gauche ou à droite.
- La recherche est fructueuse si la valeur est trouvée en un nœud ; infructueuse si elle aboutit à un sous-arbre vide.
- Algorithme :

```
fonction etqPrésente (ABR A, Étiquette E) → Booléen
  si estVide(A) alors
    retourner Faux
  sinon
    si racine(A)=E alors
      retourner Vrai
    sinon
      si racine(A)>E alors
        retourner etqPrésente (gauche(A), E)
      sinon
        retourner etqPrésente (droit(A), E)
      finsi
    finsi
  finsi
```

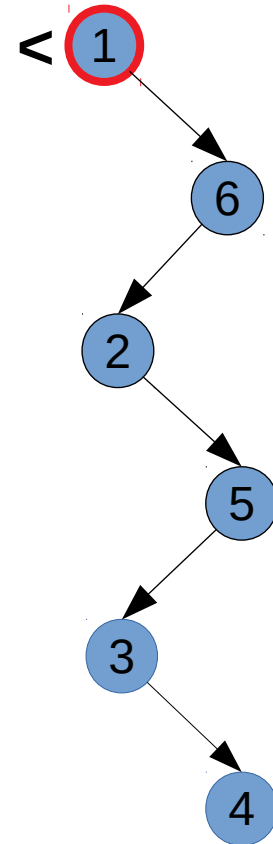
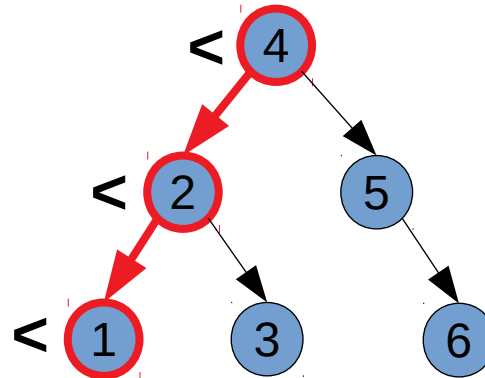
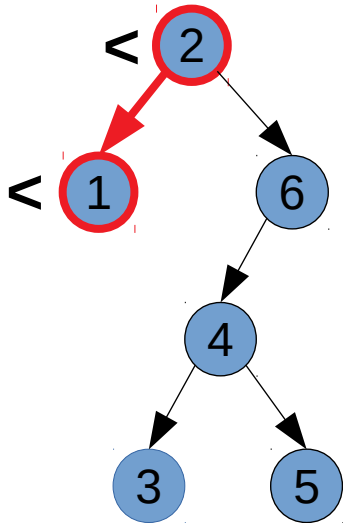
Recherche dans un ABR (2)

- Exemple : $\text{etqPrésente}(A, 5) \rightarrow \text{Vrai}$



Recherche dans un ABR (3)

- Exemple : $\text{etqPrésente}(A,0) \rightarrow \text{Faux}$



Recherche dans un ABR (4)

Propriétés : *(justification intuitive)*

- Terminaison
 - On passe d'un arbre à un sous-arbre \Rightarrow ça converge vers l'arbre vide
- Correction
 - En chaque nœud, les 3 cas couvrent toutes les possibilités
- Complexité :
 - On suit un chemin de la racine à la valeur cherchée, ou (un fils manquant d')une feuille au pire.
 - \Rightarrow En meilleur cas, la valeur est trouvée dans la racine : $\Omega(1)$
 - \Rightarrow En pire cas, on va jusqu'à la feuille la plus profonde : $O(H)$ où H est la hauteur de l'ABR

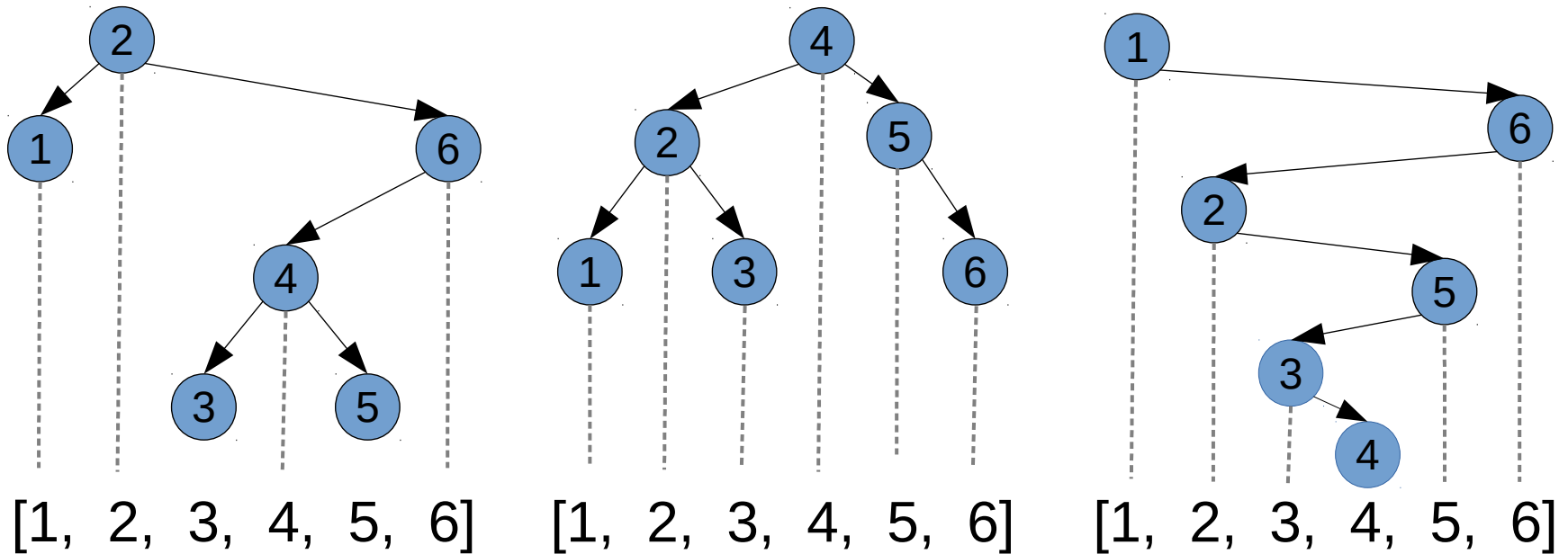
Recherche dans un ABR (5)

Analyse plus fine de la complexité :

- La recherche prend un temps $O(H)$, mais que vaut H par rapport au nombre de nœuds de l'ABR ?
 - Rappel (séance 1) : $\log_2(N+1)-1 \leq H \leq N-1$
- Donc l'algorithme est en $O(N)$ en « pire des pire » cas : l'ABR est « dégénéré » en une liste (cf exemple de droite)
 - Équivaut à une recherche classique dans une liste quelconque
- Mais en $O(\log(N))$ pour un ABR bien « équilibré » (cf. exemple du milieu)
 - Équivaut à une recherche dichotomique dans une liste triée

Recherche dans un ABR (6) : conclusion

On peut mettre en relation la structure de l'ABR avec l'organisation d'une liste triée :



⇒ Faire une recherche dans un ABR, c'est faire une recherche dans une liste triée en sautant aux indices correspondant : ABR équilibré \Leftrightarrow recherche dichotomique

Ajout dans un ABR (1)

- L'intérêt des ABR vis-à-vis des listes se trouve dans l'efficacité des opérations de modification
 - Dans une liste : $O(N)$
 - Dans un ABR : $O(H)$, c'est à dire $O(\log(N))$ si équilibré
- Principe de l'ajout :
 - Pour que l'élément ajouté soit retrouvé lors d'une future recherche, il faut l'insérer à l'endroit où conduira cette recherche
 - Or la recherche suit un chemin unique dans l'ABR !
 - Donc on va faire une recherche, et insérer le nouveau nœud lorsqu'on aboutit à un « fils manquant »

Ajout dans un ABR (2)

- Algorithme :

```
fonction ajouter(ABR A, Étiquette E)
  si estVide(A) alors
    A ← embranche(E, arbreVide(), arbreVide())
  sinon
    si racine(A) ≥ E alors
      ajouter(gauche(A), E)
    sinon
      ajouter(droit(A), E)
    finsi
  finsi
```

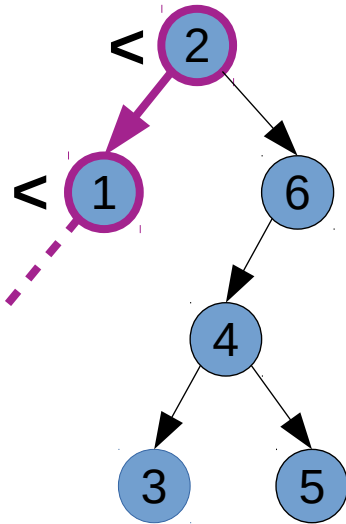
⚡ **Attention !** Cet algorithme ne marche qu'en considérant que :

- les paramètres sont passés en « partage mémoire » (modifier la valeur d'un paramètre aura un effet dans le programme appelant)
- gauche et droit donnent accès en modification aux sous-arbres (on pourrait écrire gauche(A) ← B par exemple)

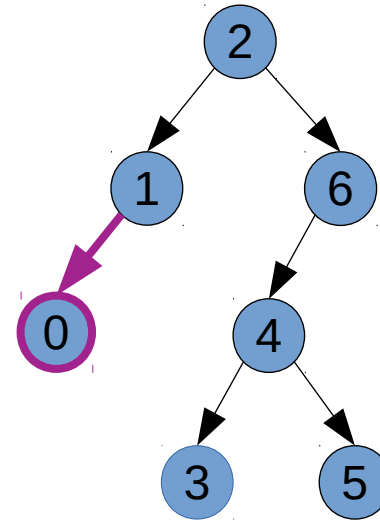
Ajout dans un ABR (3)

- Exemple 1 : ajouter(A,0)

étape 1 : recherche



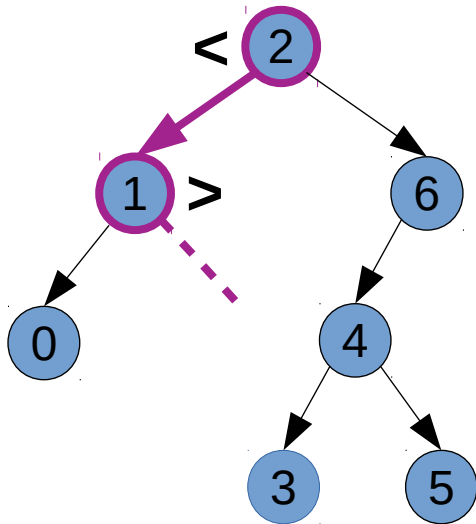
étape 2 : insertion



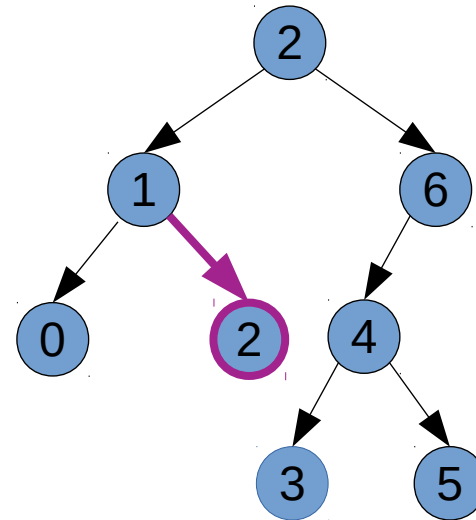
Ajout dans un ABR (4)

- Exemple 2 : ajouter(A,2)

étape 1 : recherche



étape 2 : insertion



- Même en cas d'égalité, on descend toujours jusqu'à un « fils manquant » (insertion d'une feuille)

Ajout dans un ABR (5)

- Propriétés :
 - Terminaison : la recherche termine, l'insertion correspond au cas de base (ni répétitive ni appel récursif)
 - Correction : la recherche est correcte (en ignorant les doublons croisés)
 - Complexité : insertion d'une feuille
 - ⇒ Au mieux $\Omega(1)$: fils manquant de la racine
 - ⇒ Au pire $O(H)$: fils manquant de la feuille la plus profonde
- Donc $O(N)$ au pire du pire (ABR dégénéré),
Mais $O(\log(N))$ avec un peu de chance (ABR équilibré)

Exercices (60')

- Proposer une analyse et un algorithme pour les problèmes suivants :
 - Recherche du minimum/maximum dans un ABR
 - Recherche du suivant de l'étiquette racine d'un sous-arbre donné
 - Recherche de la k-ième plus petite étiquette dans un ABR

Éléments de solution : recherche du minimum

- Principe : il suffit de partir systématiquement à gauche depuis la racine ; le dernier nœud atteint avant un « fils manquant » est le minimum (*le justifier/démontrer*)
- Algorithme (*une version itérative pour changer*) :

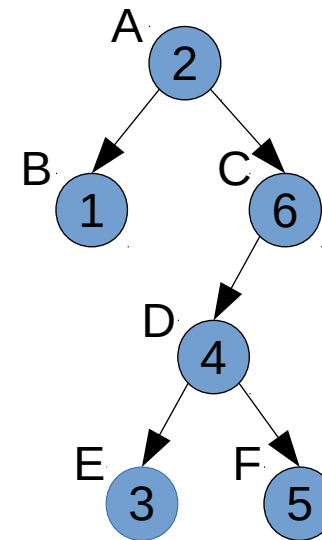
```
fonction minimum(ABR A) → Étiquette  
  variable ABR a  
  a ← A // copie pour éviter la modification du paramètre  
  tant que non estVide(gauche(a)) faire  
    a ← gauche(a)  
  fintq  
  retourner racine(a)
```

Et pour le maximum ?

- Précondition : A non vide
- Terminaison : gauche(a) « plus petit » que a
- Correction : cf. démonstration supra
- Complexité : $\Omega(1)$, $O(H)$, comme une recherche quelconque

Éléments de solution : recherche du suivant

- Principe : Le suivant d'une étiquette en racine d'un sous-arbre B dans un ABR A est
 - Soit le minimum du sous-arbre droit de B s'il existe
 - Soit le premier *ancêtre* de B *par la gauche*
- Un exemple pour comprendre :
 - $\text{suivant}(A) \rightarrow E$
= $\text{minimum}(C)$
 - $\text{suivant}(F) \rightarrow C$
= $\text{ancêtreG}(F)$
 - **$\text{suivant}(C) ?$**
- Pour écrire cet algorithme, il faut une opération de plus :
père(A) : retourne le *sur-arbre* dont A est le sous-arbre gauche ou droit



Et pour le précédent ?

Éléments de solution : $k^{\text{ème}}$ plus petite étiquette (1)

- Principe :

- On peut partir du nœud minimum et passer $k-1$ fois au suivant. Ce n'est pas une solution très efficace (*justifier*)
- En stockant une information de plus en chaque nœud, on peut faire mieux (compromis espace-temps) : Si chaque nœud connaît le nombre de nœuds du sous-arbre dont il est racine, on peut savoir s'il faut partir à gauche ou à droite :
 - à gauche si $\text{nbNoeuds}(\text{gauche}(A)) \geq k$;
 - à droite autrement, mais en prenant soin d'actualiser k (lui retirer le nombre de nœuds « esquivés » à gauche) ;
 - et quand $\text{nbNoeuds}(\text{gauche}(A)) = k-1$, on est arrivé !

⇒ on considère l'opération supplémentaire **nbNoeuds(A)** qui donne le nombre de nœuds dans l'ABR A en temps constant

Éléments de solution : $k^{\text{ème}}$ plus petite étiquette (2)

- Algorithme (*version itérative*) :

- fonction kiemeÉtiquette(ABR A, Entier k) → Étiquette
 variable ABR a ; Entier nG
 a ← A // copie pour éviter la modification du paramètre
 nG ← nbNœuds(gauche(a)) // 0 si gauche(a) est vide
 tant que nG ≠ k-1 faire
 si nG ≥ k alors
 a ← gauche(a)
 sinon
 a ← droit(a)
 k ← k - nG - 1 // nœuds à gauche et racine
 finsi
 nG ← nbNœuds(gauche(a)) // 0 si gauche(a) est vide
 fintq
 retourner racine(a)

- Précondition : $1 \leq k \leq \text{nbNoeuds}(A)$
- Terminaison : nG+1-k peut servir de variant (*le démontrer*)
- Correction partielle : difficile ! On se contentera de l'intuition donnée au transparent précédent
- Complexité : $O(H)$

Arbres de recherche : conclusion

- Les arbres binaires de recherche offrent une alternative intéressante aux listes pour gérer une collection de données « requêtable » :
 - on peut rechercher une étiquette en temps logarithmique,
 - tout en ajoutant/retirant des éléments en temps logarithmique ... à condition que l'arbre soit équilibré !

⇒ De nombreuses variantes « auto-équilibrées » ont été proposées : AVL, arbres rouge-noir, ... (*hors programme NSI !*)
- L'ABR est ainsi une structure appropriée pour implémenter un dictionnaire ! Mais Python utilise une autre structure efficace : la table de hachage (*hors programme NSI*)