7.5 Algorithmes sur les arbres binaires de recherche

7.5.1 Afficher les étiquettes d'un arbre binaire de recherche dans l'ordre croissant

Comme nous l'avons vu dans l'exercice précédent, c'est la parcours infixe qui permet un affichage dans l'ordre croissant (ou l'ordre alphabétique).

7.5.2 Recherche dans un ABR - Version récursive

L'algorithme

L'intérêt des ABR est d'être plus efficace dans la recherche d'une valeur.
En effet, pour rechercher une valeur, il suffit de la comparer avec la valeur à la racine, puis, si elle est différente, poursuivre la recherche dans un seul sous-arbre.

Pour rechercher une clé dans un arbre binaire de recherche, on peut d'abord la comparer avec la racine. Si la clé est présente à la racine, on renvoie Vrai. Si la clé est inférieure à la racine, on cherche la clé dans le sous-arbre de gauche. Si la clé est supérieure à la racine, on cherche alors dans le sous-arbre de droite. Si la clé n'a pas été trouvée, on retourne Faux.

Exercice 7.36

Écrire une fonction `recherche(tree,elt)` qui prend pour argument `tree` un arbre binaire de recherche et un `elt`, et qui retourne `False` ou `True` suivant la présence ou non de `elt` dans l'arbre `tree`.***

```
def recherche(tree,elt):
```

Efficacité de la recherche

Si l'arbre est équilibré : si les éléments sont a peu près bien répartis entre les deux sous-arbres, la fonction recherche élimine à chaque appel récursif la moitié des éléments !

Approche

Activités qui permettent de déterminer la complexité de l'algorithme précédent.

1. Compléter le tableau suivant :

Nombre n en base 10	Troncature de $log_2(n)$	n en base 2	Nb de chiffres de l'écriture binai
15			
28			
100			
90			
64			
2			

☞ Le nombre de chiffre(s) de l'écriture bianaire du nombre n_{10} est $log_2(n)+1$.

2. Considérons maintenant un nombre n écrit en binaire.

n_{10}	n_2	$\frac{n}{2}$ base 10 à 1 près	$\frac{n}{2}$ base 2 à 1 près
100			
50			
25			
12			
6			
3			
1			

☞ Pour diviser un nombre binaire par deux, il suffit d'enlever le bit de poids faible (à un près)

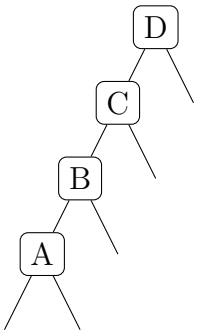
De ces deux tableaux, on peux en déduire que si la taille de l'arbre est n, on aura (si l'arbre est équilibré) dans le pire des cas $log_2(n)+1$ étapes.

Propriété

La compléxité de la recherche d'un élément dans le cas d'un arbre équilibré est en $O(log_2(n))$.

☛ En conclusion :
Considérons un arbre binaire de recherche, noté `T`, dont la taille est n (ce qui signifie que le nombre de noeuds est égal à n).
Si, comme supposé, les éléments sont répartis à peu près équitablement entre les sou-arbres, la fonction `recherche_rec_abr` élimine environ la moitié des éléments à chaque étape.
Considérons m l'écriture de n en base 2. A chaque étape, m étant divisé par 2, m va perdre un chiffre (celui de droite). L'algorithme se termine (dans le pire des cas) lorsque m n'a plus qu'un chiffre.
Ainsi, le nombre d'étapes nécessaires est égal au nombre de chiffres de m, soit $log_2(n)+1$.
Autrement dit, pour un arbre de taille n, il faut environ $log_2(n)$ étapes.
On peut ainsi en déduire que $O(n) = log_2(n)$

Si l'arbre n'est pas équilibré : Prenons un exemple :



Dans le pire des cas, l'arbre peut être complétement déséquilibré ; c'est le cas de l'exemple où on est en présence d'un **peigne**. Sa hauteur est ici égale ou nombre de nœuds.
Dans ce cas, notre algorithme n'est pas très efficace, car il est suceptible de parcourir l'arbre en entier. Dans ce cas, l'algorithme n'est pas meilleur que celui recherchant une valeur sans une

liste chaînée.

D'une façon générale , le coût de la recherche dans un ABR dépend de sa hauteur ; plus précisémment, ce coût est majoré par sa hauteur. En effet, à chaque étape, on descend dans le sous-arbre droite ou le sous-arbre gauche, et on ne peut donc pas répéter ceci un nombre de fois supérieur à sa hauteur.

7.5.3 Recherche dans un ABR - Version itérative

Exercice 7.37
Créer une fonction `recherche_iteratif_abr(tree,elt)` avec `tree` un ABR et `elt` un `int` ou `str`. La fonction renvoie `True` ou `False` en fonction du résultat de la recherche

7.5.4 Recherche du minimum dans un ABR

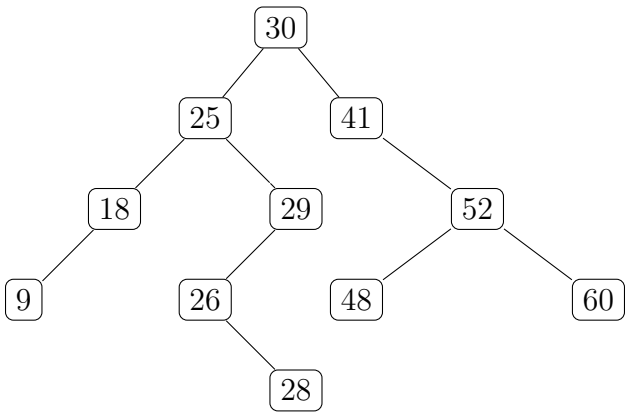
Exercice 7.38
Construire une fonction **min(tree)** qui accepte en argument un ABR et qui renvoie la clé minimale

7.5.5 Recherche du maximum dans un ABR

Exercice 7.39
Construire une fonction **max(tree)** qui accepte en argument un ABR et qui renvoie la clé maximale.

7.5.6 Insertion d'une clé dans un arbre binaire de recherche

☞ Insertions aux feuilles : cela signifie que l'on suppose que element va trouver sa place au niveau des feuilles.



On souhaite inserer la clé "23" dans l'ARB précédent.
☞ Combien y a-t-il de places possibles ?
☞ Quelle instruction permet d'effectuer cet ajout ?

Exercice 7.40

On souhaite maintenant créer une fonction `ajoute(x,tree)` qui prend en argument un ABR et une clé, et modifie en place l'ABR, en ajoutant la clé au bon endroit.