Arbres binaires de recherche

1. Définition et opérations

Définition: Un arbre binaire de recherche (ou dictionnaire binaire ou arbre binaire trié) est un arbre binaire vérifiant la propriété suivante : soient *x* et *y* deux nœuds de l'arbre, si *y* est un nœud du sous-arbre gauche de *x*, alors *étiquette(y)* $\leq$ *étiquette(x)*, si *y* est un nœud du sous-arbre droit de *x*, alors *étiquette(y)* > *étiquette(x)*.

Il résulte de cette définition que le parcours infixe d'un tel arbre produit une suite d'étiquettes en ordre croissant.

On utilise les arbres binaires de recherche pour effectuer des recherches rapides : on insère systématiquement dans le sous-arbre gauche les éléments d'étiquette plus petite ou égale à celle de la racine et dans le droit ceux qui ont une étiquette plus grande. L'insertion d'un nouveau nœud peut s'effectuer à une feuille, sans trop de bouleversement, ou à la racine, avec plus de conséquences.

La figure 1 présente deux exemples d'arbres binaires de recherche. Bien que différents, ces deux arbres contiennent exactement les mêmes valeurs.

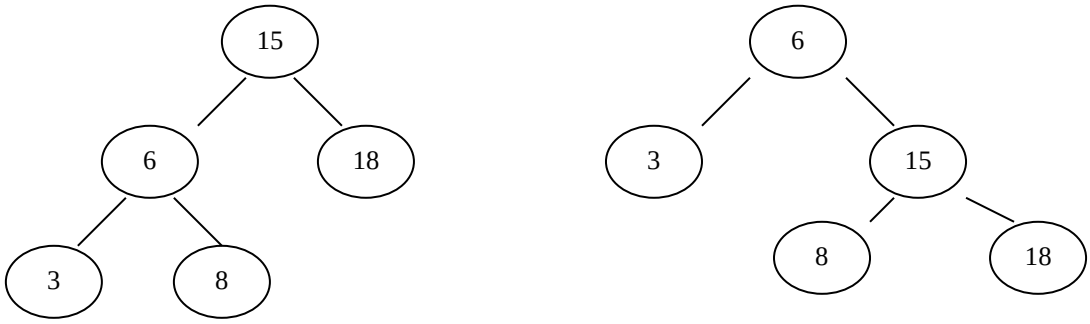


Figure 1: deux exemples d'arbres binaires de recherche.

Voici l'exemple de l'arbre de la figure 1 (gauche) après insertion de 7 en feuille et à la racine:

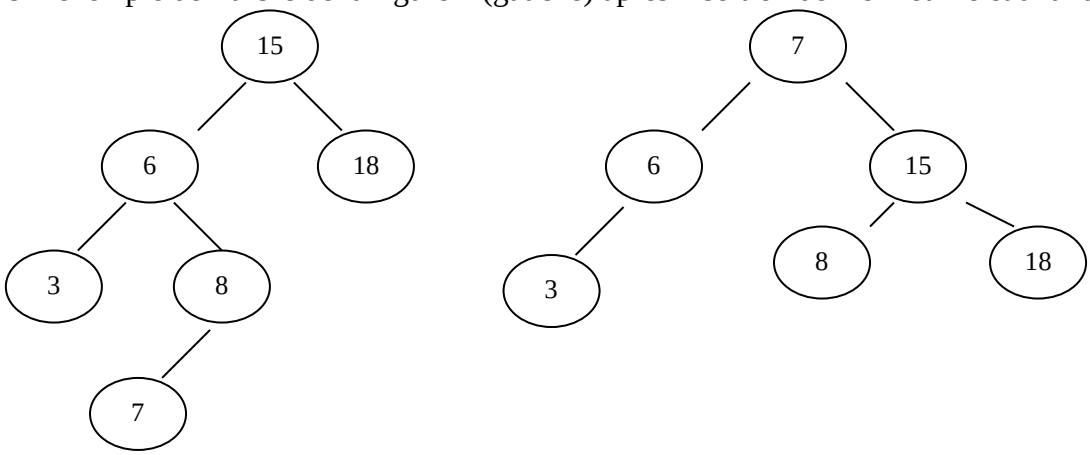


Figure 2: Insertion de la valeur 7 en feuille (gauche) et à la racine (droite).

Nous spécifions les opérations des arbres binaires de recherche. On suppose dans cette spécification que la même étiquette ne figure pas deux fois dans l'arbre. Opérations sur les arbres binaires:

arbrevide	:		→	abin
enrac	:	T • abin • abin	→	abin
ag, ad	:	abin	→	abin
insf	:	T • abin	→	abin // insertion à une feuille
insr	:	T • abin	→	abin // insertion à la racine
sup	:	T • abin	→	abin
min,max	:	abin	→	T
rech:	:	T • abin	→	abin
hauteur	:	abin	→	entier

Par rapport à la spécification des arbres binaires généraux, nous ne définissons ici que les générateurs qui imposent des contraintes plus fortes.

Pré-conditions :
Pré(enrac(x,a1,a2)) ≡ x≠ω et (non vide(a1) ⊃ max(a1)≤x) et (non vide(a2) ⊃ x<max(a2))
Pré(min(a))=**Pré**(max(a)) ≡ non vide(a)

Axiomes :
min(enrac(x,a1,a2)) = si vide(a1) alors x sinon min(a1)
max(enrac(x,a1,a2)) = si vide(a2) alors x sinon max(a2)

rech(arbrevide,x) = arbrevide
rech(enrac(y,a1,a2),x) = si x=y alors enrac(y,a1,a2)
sinon si x < y alors rech(a1,x) sinon rech(a2,x)

insf(arbrevide,x) = enrac(x, arbrevide, arbrevide)
insf(enrac(y,a1,a2),x) = si x=y alors enrac(y,a1,a2)
sinon si x<y alors enrac(y, insf(a1,x),a2)
sinon enrac(y, a1, insf(a2,x))

Description des opérations et de la structure de données:

```
type abin = {
     T         etiq;
     abin      *ag, *ad;
}
// Création d'un arbre vide
abin *arbrevide()
{
     renvoyer NULL
}
```

// créer un arbre à partir du fils gauche et droit
```
abin  *enraciner(T x, abin *g, abin *d)
{
      abin *nouv := nouveau abin;
      nouv->etiq:=x;
      nouv->ag:=g;
      nouv->ad:=d;
}
```

// recherche du min et du max
```
T     min(abin *a)
{
      si a == NULL alors // erreur
      si a->ag == NULL alors renvoyer a->etiq
      sinon renvoyer min(a->ag);
}
T     max(abin *a)
{
      si a == NULL alors // erreur
      si a->ad == NULL alors renvoyer a->etiq
      sinon renvoyer max(a->ad);
}
```

// recherche d'un élément
```
abin  *rech(T x, abin *a)
{
      si a == NULL alors renvoyer a
      si a->etiq == x alors renvoyer a
      si a->etiq < x alors renvoyer rech(x, a->ad);
      sinon  renvoyer rech(x, a->ag);
}
```

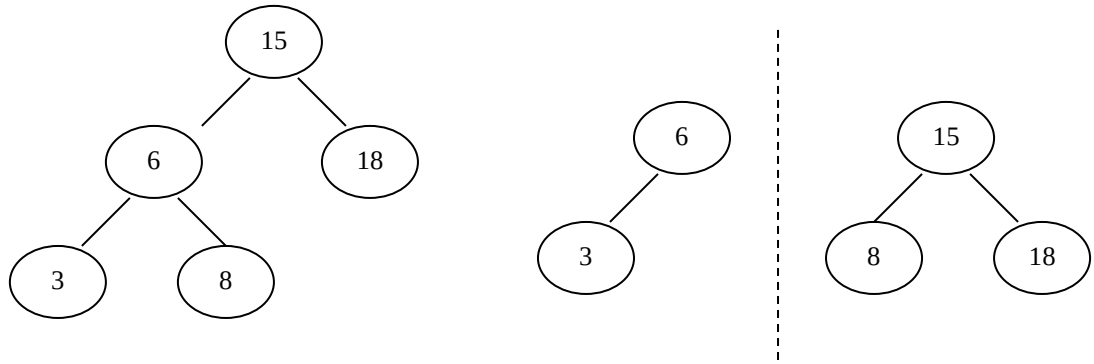
Pour insérer en feuille, on parcourt l'arbre comme pour la recherche jusqu'à ce que l'emplacement soit trouvé:

// insertion à la feuille
```
abin  *insf(T x, abin *a)
{
      si a == NULL alors
            renvoyer enraciner(x, arbrevide(),arbrevide())
      si a->etiq < x alors a->ad := insf(x, a->ad)
      sinon si a->etiq > x alors  a->ag := insf(x, a->ag)
      renvoyer a
}
```

Pour insérer à la racine, c’est un peu plus difficile.

On utilise une fonction *coupure*. Cette fonction prend en entrée une étiquette *x* et un arbrebinaire de recherche *a* et retourne deux arbres binaires de recherche contenant tous les éléments de *a*, et tels que toute étiquette du premier soit inférieure ou égale à x et toute étiquette du second soit supérieure à x:

Exemple: coupure de l'arbre de la figure 1, selon x=7



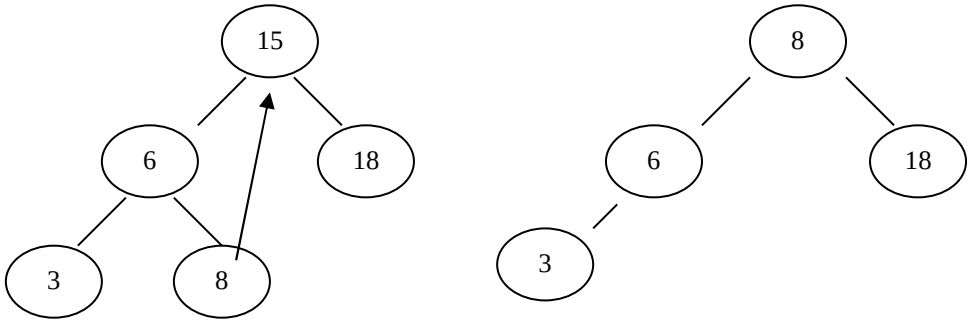
On procède par récursivité: on suppose que l'on sait couper un arbre, puis on résout le cas de l'arbre vide (condition de terminaison) et on applique la coupure aux sous arbres gauche ou droit en fonction de la valeur de x. La fonction suivante réalise l'opération de coupure et renvoie dans a1 et a2 le résultat de la coupure par effet de bord (d'où le pointeur double).

```
// coupure
coupure(T x, abin *a, abin **a1, abin **a2)
{
      abin *sousg, *sousd

      si a == NIL alors
            *a1 := arbrevide()
            *a2 := arbrevide()
      sinon si x <= a->etiq alors
            coupure(x, a->ag, &sousg, &sousd)
            *a1 := sousg
            a->ag := sousd
            *a2 := a
      sinon
            coupure(x, a->ad, &sousg, &sousd)
            a->ad := sousg
            *a1 := a
            *a2 := sousd
}
```

L'opération d'insertion à la racine ***insr*** s'écrit trivialement à l'aide de ***coupure***:

```
// insertion en racine
abin  *insr(T x, abin *a)
{
      abin *sousg, *sousd

      coupure(x, a, &sousg, &sousd)
      renvoyer enraciner(x, sousg, sousd)
}
```

Pour supprimer un élément d'un arbre binaire de recherche, il faut d'abord trouver cet élément, puis le remplacer par son prédécesseur. Le prédécesseur (à ne pas confondre avec le précédent dans une liste chainée) est la valeur dans l'arbre *immédiatement plus plus*. Cela revient à rechercher le max dans le sous arbre gauche. En effet, le sous arbre gauche contient toutes les valeurs plus petites, dont fait également partie son prédécesseur. Celui-ci est le plus grand de l'ensemble des plus petits. Le schéma suivant présente le principe de suppression par remplacement. On supprime la valeur 15, qui se trouve à la racine. Le max du sous arbre gauche est 8, qui correspond à son prédécesseur. On retire donc la feuille 8, que l'on met à la place de 15.



Nous allons introduire une opération *otermax()* qui permet de retirer le maximum d'un arbre, tout en renvoyant ce maximum.

```
// retirer le maximum
abin *otermax(abin *a, T *max)
{
      abin *temp

      si a==NIL alors // erreur
      si a->ad == NIL alors
            *max = a->etiq
            temp := a->ag
            detruire a
            renvoyer temp
      sinon
            a->ad := otermax(a->ad, max)
}
```

La suppression d'un élément se fait en remplaçant cet élément pas son prédécesseur, c'est à dire le maximum de tous les éléments inférieurs:

```
// retirer le maximum
abin *supprimer(T x, abin *a)
{
      T max;
      abin *temp=a;

      si a==NULL alors renvoyer a
      si a->etiq == x alors
            si a->ag == NULL alors temp:=a->ad; detruire a
            sinon si a->ad == NULL alors temp:=a->ag; detruire a
```

```
            sinon a->ag := otermax(a->ag, &max); a->etiq:= max;
      sinon si a->etiq < x alors a->ad := supprimer(x,a->ad)
      sinon a->ag := supprimer(x, a->ag)
      renvoyer temp
}
```

L'analyse de la complexité de ces algorithmes se ramène à l'étude de certaines caractéristiques des arbres binaires. On sait que la profondeur et la profondeur moyenne d'un arbre binaire contenant *n* nœuds est une quantité comprise entre $log_2(n)$ et *n*.

Complexité : tous les algorithmes que nous avons vus sont en $O(h)$ si *h* est la hauteur de l'arbre. Le cas $h=log_2(n)$ se présente si l'arbre binaire est parfaitement équilibré. Nous allons donc étudier ce type d'arbre par la suite.