

# Arbre binaire de recherche

Quentin Fortier

January 4, 2022

# Arbre binaire de recherche (ABR)

## Définition

Un **arbre binaire de recherche** (ABR ou BST en anglais) est un arbre binaire tel que, pour chaque noeud d'étiquette  $r$  et de sous-arbres  $g$  et  $d$ ,  $r$  est supérieur à toutes les étiquettes de  $g$  et inférieur à toutes les étiquettes de  $d$ .

Il faut que les étiquettes soient comparables (des nombres par exemple).

# Arbre binaire de recherche (ABR)

## Définition

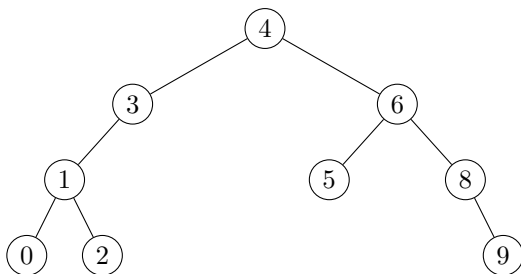
Un **arbre binaire de recherche** (ABR ou BST en anglais) est un arbre binaire tel que, pour chaque noeud d'étiquette  $r$  et de sous-arbres  $g$  et  $d$ ,  $r$  est supérieur à toutes les étiquettes de  $g$  et inférieur à toutes les étiquettes de  $d$ .

Il faut que les étiquettes soient comparables (des nombres par exemple).

Un ABR est donc une structure « triée » permettant de généraliser la recherche par dichotomie dans un tableau trié.

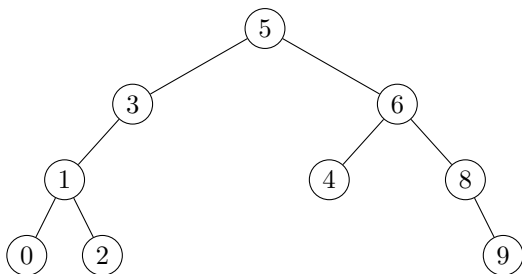
# Arbre binaire de recherche (ABR)

Exemple d'ABR :



# Arbre binaire de recherche (ABR)

Exemple d'arbre qui n'est pas un ABR :



# Opérations sur les ABR

Opérations sur les ABR :

- has : test d'appartenance
- add : ajout d'un élément
- del : supprimer un élément

# Opérations sur les ABR

Opérations sur les ABR :

- has : test d'appartenance
- add : ajout d'un élément
- del : supprimer un élément

Ces opérations doivent conserver la structure d'ABR et se feront en  $O(h)$ , où  $h$  est la hauteur.

# Opérations sur les ABR

Opérations sur les ABR :

- has : test d'appartenance
- add : ajout d'un élément
- del : supprimer un élément

Ces opérations doivent conserver la structure d'ABR et se feront en  $O(h)$ , où  $h$  est la hauteur.

Dans un arbre binaire (non ABR), has demande une complexité linéaire en le nombre  $n$  de noeuds.

Dans le meilleur des cas,  $h = O(\log(n))$  et has est beaucoup plus rapide avec un ABR.



# Opérations sur les ABR : Test d'appartenance

---

```
1  bool has(int e, tree* t) {  
2      if(!t)  
3          return false;  
4      if(t->elem == e)  
5          return true;  
6      if(e < t->elem)  
7          return has(e, t->g);  
8      return has(e, t->d);  
9  }
```

---

Complexité :

# Opérations sur les ABR : Test d'appartenance

---

```
1  bool has(int e, tree* t) {  
2      if(!t)  
3          return false;  
4      if(t->elem == e)  
5          return true;  
6      if(e < t->elem)  
7          return has(e, t->g);  
8      return has(e, t->d);  
9  }
```

---

Complexité :  $O(h)$ , où  $h$  est la hauteur de  $t$ , car il faut parcourir une branche

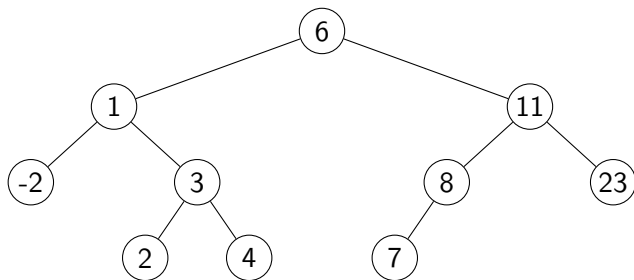
# Opérations sur les ABR : Ajout

---

```
1  let rec add e = function
2      | E -> N(e, E, E)
3      | N(r, g, d) when e < r -> N(r, add e g, d)
4      | N(r, g, d) -> N(r, g, add e d)
```

---

Exemple : ajouter 10 dans l'arbre suivant :



# Opérations sur les ABR : Ajout

---

```
1  tree* add(int e, tree* t) {  
2      if(!t)  
3          return new_node(e);  
4      if(e < t-> elem)  
5          add(e, t->g);  
6      else  
7          add(e, t->d);  
8      return t;  
9  }
```

---

# Opérations sur les ABR : Ajout

---

```
1  tree* add(int e, tree* t) {  
2      if(!t)  
3          return new_node(e);  
4      if(e < t-> elem)  
5          add(e, t->g);  
6      else  
7          add(e, t->d);  
8      return t;  
9  }
```

---

On pourrait aussi ne pas renvoyer de valeur, mais il faudrait alors un double pointeur pour traiter le cas vide (**NULL**) :

```
void add(int e, tree** t)
```

## Opérations sur les ABR : Suppression

Fonction utilitaire : calculer et supprimer le maximum d'un ABR.

# Opérations sur les ABR : Suppression

Fonction utilitaire : calculer et supprimer le maximum d'un ABR.

Il faut chercher toujours à droite :

---

```
1  let rec del_max = function
2      | E -> max_int, E
3      | N(r, g, E) -> r, g
4      | N(r, g, d) -> let m, d' = del_max d in
5                          m, N(r, g, d')
```

---

# Opérations sur les ABR : Suppression

Pour supprimer un noeud d'étiquette  $e$  :

- 1 Chercher un noeud  $N(x, g, d)$  (comme pour `has`)
- 2 Remplacer  $x$  par la maximum de  $g$ , pour conserver un ABR.

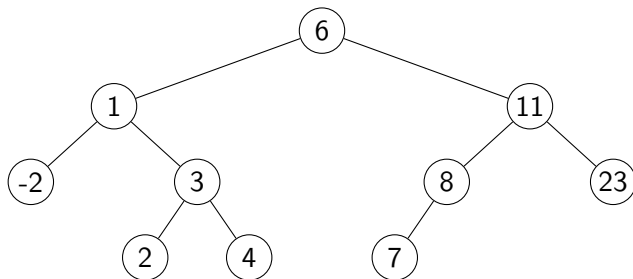


## Opérations sur les ABR : Suppression

Supprimer un élément  $e$  (ici 6) dans un ABR :

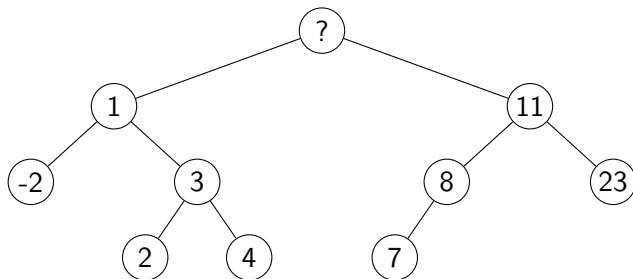
# Opérations sur les ABR : Suppression

Supprimer un élément  $e$  (ici 6) dans un ABR :



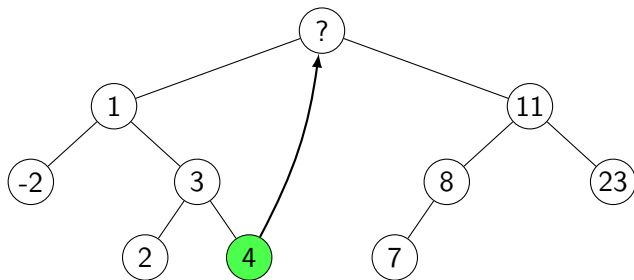
# Opérations sur les ABR : Suppression

Supprimer un élément  $e$  (ici 6) dans un ABR :



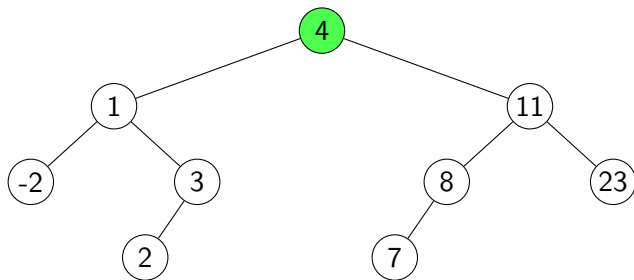
# Opérations sur les ABR : Suppression

Supprimer un élément  $e$  (ici 6) dans un ABR :



# Opérations sur les ABR : Suppression

Supprimer un élément  $e$  (ici 6) dans un ABR :



## Opérations sur les ABR : Suppression

---

```
1  let rec del e = function
2      | E -> E
3      | N(r, g, d) when e = r -> let m, g' = del_max g in
4                                  N(m, g', d)
5      | N(r, g, d) when e < r -> N(r, del e g, d)
6      | N(r, g, d) -> N(r, g, del e d);;
```

---

# Opérations sur les ABR : Suppression

---

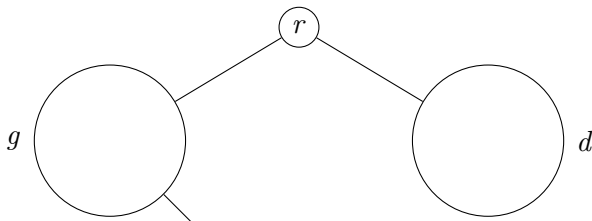
```
1  let rec del e = function
2      | E -> E
3      | N(r, g, d) when e = r -> let m, g' = del_max g in
4                                  N(m, g', d)
5      | N(r, g, d) when e < r -> N(r, del e g, d)
6      | N(r, g, d) -> N(r, g, del e d);;
```

---

Complexité :  $O(h)$  ()

# Opérations sur les ABR : Suppression

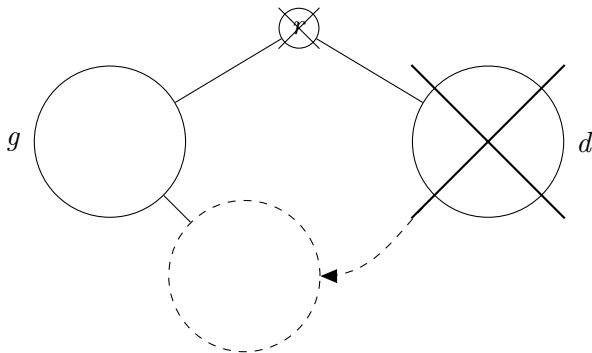
Autre façon de supprimer un élément : fusionner les deux sous-arbres restants.





# Opérations sur les ABR : Suppression

Autre façon de supprimer un élément : fusionner les deux sous-arbres restants.



# Opérations sur les ABR : Suppression

---

```
1  let rec fusion g d = match g with
2    | E -> d
3    | N(gr, gg, gd) -> N(gr, gg, fusion gd d);;
4
5  let rec del e = function
6    | E -> E
7    | N(r, g, d) when e = r -> fusion g d
8    | N(r, g, d) when e < r -> N(r, del e g, d)
9    | N(r, g, d) -> N(r, g, del e d);;
```

---

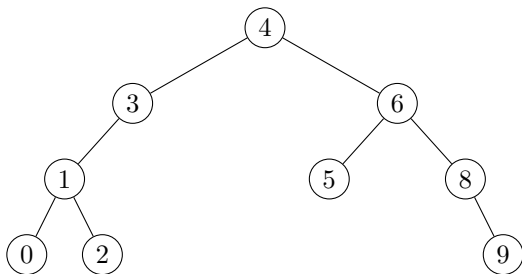
# Opérations sur les ABR : Suppression

## Exercice

Refaire toutes les fonctions d'ABR en C.

# Tri avec un ABR

Le parcours infixe d'un ABR donne un tri :



Parcours infixe : 0, 1, 2, 3, 4, 5, 6, 8, 9

# Tri avec un ABR

## Exercice

Écrire une fonction qui trie une liste en construisant un ABR puis en renvoyant son parcours infixe.

## Exercice

Écrire une fonction qui trie une liste en construisant un ABR puis en renvoyant son parcours infixe.

---

```
1  let tri_abr l =  
2      List.fold_right add E l  
3      |> infixe
```

---