Chapitre 1 : Les arbres (saison 2)

Option Informatique – MP

Lycée Chateaubriand

Table des matières

Plan

Implémentation

Arbres binaires de recherche

Dictionnaires

Dictionnaire

Définition 1 (Dictionnaire)

La structure de données abstraite de dictionnaire consiste en un ensemble de couples (clés, données) tels que les clés soient dans un ensemble totalement ordonné (la plupart du temps des entiers). On dispose des opérations suivantes :

-

Dictionnaire

Définition 1 (Dictionnaire)

La structure de données abstraite de dictionnaire consiste en un ensemble de couples (clés, données) tels que les clés soient dans un ensemble totalement ordonné (la plupart du temps des entiers). On dispose des opérations suivantes :

- *Créer un dictionnaire vide*
-

Dictionnaire

Définition 1 (Dictionnaire)

La structure de données abstraite de dictionnaire consiste en un ensemble de couples (clés, données) tels que les clés soient dans un ensemble totalement ordonné (la plupart du temps des entiers). On dispose des opérations suivantes :

- *Créer un dictionnaire vide*
- *Insertion : on insère un nouveau couple*
-

Dictionnaire

Définition 1 (Dictionnaire)

La structure de données abstraite de dictionnaire consiste en un ensemble de couples (clés, données) tels que les clés soient dans un ensemble totalement ordonné (la plupart du temps des entiers). On dispose des opérations suivantes :

- *Créer un dictionnaire vide*
- *Insertion : on insère un nouveau couple*
- *Chercher une clé et si elle est présente on renvoie la donnée.*
-

Dictionnaire

Définition 1 (Dictionnaire)

La structure de données abstraite de dictionnaire consiste en un ensemble de couples (clés, données) tels que les clés soient dans un ensemble totalement ordonné (la plupart du temps des entiers). On dispose des opérations suivantes :

- *Créer un dictionnaire vide*
- *Insertion : on insère un nouveau couple*
- *Chercher une clé et si elle est présente on renvoie la donnée.*
- *Suppression d'un couple*

Dictionnaire

Définition 1 (Dictionnaire)

La structure de données abstraite de dictionnaire consiste en un ensemble de couples (clés, données) tels que les clés soient dans un ensemble totalement ordonné (la plupart du temps des entiers). On dispose des opérations suivantes :

- *Créer un dictionnaire vide*
- *Insertion : on insère un nouveau couple*
- *Chercher une clé et si elle est présente on renvoie la donnée.*
- *Suppression d'un couple*

Remarque : Dans la majorité des cas, on demande à ce qu'un même clé n'apparaissent pas deux fois dans le dictionnaire.

L'exemple le plus classique est le dictionnaire usuel.

Implémentation

Regardons deux implémentations naïves :

- Une liste de couples. On peut regarder la complexité des opérations que l'on veut réaliser par rapport à la taille du dictionnaire.
  - La recherche :

Implémentation

Regardons deux implémentations naïves :

- Une liste de couples. On peut regarder la complexité des opérations que l'on veut réaliser par rapport à la taille du dictionnaire.
  - La recherche : linéaire
  - L'insertion :

Implémentation

Regardons deux implémentations naïves :

- Une liste de couples. On peut regarder la complexité des opérations que l'on veut réaliser par rapport à la taille du dictionnaire.
  - La recherche : linéaire
  - L'insertion : constant
  - La suppression :

Implémentation

Regardons deux implémentations naïves :

- Une liste de couples. On peut regarder la complexité des opérations que l'on veut réaliser par rapport à la taille du dictionnaire.
  - La recherche : linéaire
  - L'insertion : constant
  - La suppression : linéaire

Implémentation

Regardons deux implémentations naïves :

- Une liste de couples. On peut regarder la complexité des opérations que l'on veut réaliser par rapport à la taille du dictionnaire.
  - La recherche : linéaire
  - L'insertion : constant
  - La suppression : linéaire
- Un tableau de couples ordonnés par les clés. Là encore, on peut regarder les complexité :
  - La recherche :

Implémentation

Regardons deux implémentations naïves :

- Une liste de couples. On peut regarder la complexité des opérations que l'on veut réaliser par rapport à la taille du dictionnaire.
  - La recherche : linéaire
  - L'insertion : constant
  - La suppression : linéaire
- Un tableau de couples ordonnés par les clés. Là encore, on peut regarder les complexité :
  - La recherche : logarithmique
  - L'insertion :

Implémentation

Regardons deux implémentations naïves :

- Une liste de couples. On peut regarder la complexité des opérations que l'on veut réaliser par rapport à la taille du dictionnaire.
  - La recherche : linéaire
  - L'insertion : constant
  - La suppression : linéaire
- Un tableau de couples ordonnés par les clés. Là encore, on peut regarder les complexité :
  - La recherche : logarithmique
  - L'insertion : linéaire
  - La suppression :

Implémentation

Regardons deux implémentations naïves :

- Une liste de couples. On peut regarder la complexité des opérations que l'on veut réaliser par rapport à la taille du dictionnaire.
  - La recherche : linéaire
  - L'insertion : constant
  - La suppression : linéaire
- Un tableau de couples ordonnés par les clés. Là encore, on peut regarder les complexité :
  - La recherche : logarithmique
  - L'insertion : linéaire
  - La suppression : linéaire

Exercice

- On considère un type `dico1` défini par `type 'a dico1 = (int*'a) list`. Écrire les fonctions d'insertion, de recherche d'un élément et de suppression d'une clé.

Exercice

- On considère un type `dico1` défini par `type 'a dico1 = (int*'a) list`. Écrire les fonctions d'insertion, de recherche d'un élément et de suppression d'une clé.

```
let insert (x,d) l = (x,d)::l;;
```

Exercice

- On considère un type `dico1` défini par `type 'a dico1 = (int*'a) list`. Écrire les fonctions d'insertion, de recherche d'un élément et de suppression d'une clé.

```
let insert (x,d) l = (x,d)::l;;
```

```
let rec cherche x l = match l with
    |[] -> failwith "la cle n'existe pas"
    | (c, d) :: q when c = x -> d
    | (c, d) :: q -> cherche x q;;
```

Exercice

- On considère un type `dico1` défini par `type 'a dico1 = (int*'a) list`. Écrire les fonctions d'insertion, de recherche d'un élément et de suppression d'une clé.

```
let insert (x,d) l = (x,d)::l;;
```

```
let rec cherche x l = match l with
    |[] -> failwith "la cle n'existe pas"
    | (c, d) :: q when c = x -> d
    | (c, d) :: q -> cherche x q;;
```

```
let rec supprime x l = match l with
    | [] -> []
    | (c, d) :: q when c = x -> supprime x q
    | (c, d) :: q -> (c,d) :: (supprime x q);;
```

Exercice

- On considère un type `dico2` défini par
  `type 'a dico2 = (int*'a) array`.
  Écrire les fonctions d'insertion, de recherche d'un élément et de suppression d'une clé. On fera attention au fait que les entrées du tableaux devront toujours être classées par ordre croissant des clés.
  On pourra commencer par coder une fonction `trouveIndice : int -> (int*'a) array -> (bool*int)` telle que `trouveIndice x t` renvoie le couple `(true,i)` s'il la première composante de `t.(i)` est `x` et renvoie un couple `false,i` sil la clé n'apparait pas dans le tableau. Dans ce cas, `i` sera l'indice du tableau où on pourra insérer une clé valant `x`.

Exercice

```
let trouveIndice x t =
  let n =Array.length t in
    let rec aux i j =
      if i = j then (false, i)
      else
        (let k = (i + j) / 2 in
            if fst (t.(k)) = x then (true, k)
            else if fst (t.(k)) < x then aux (k + 1) j
            else aux i k)
    in aux 0 n;;
```

Exercice

```
let trouve x t = let b, i = trouveIndice x t in
    if b then snd t.(i) else failwith "la cle n'existe pas";;
```

Exercice

```
let trouve x t = let b, i = trouveIndice x t in
    if b then snd t.(i) else failwith "la cle n'existe pas";;
```

```
let insert (x, d) t =
  let (b, i) = trouveIndice x t in
    if b then t
    else
      (let n = Array.length t in
          let res = Array.make (n + 1) (x, d) in
            for k = 0 to i-1 do
               res.(k) <- t.(k)
            done;
            for k = (i+1) to n do
            res.(k)<- t.(k-1)
            done;
            res);;
```

Exercice

```
let supprime x t =
  let (b, i) = trouveIndice x t in
    if b then
      (let n = Array.length t in
          let res = Array.make (n - 1) t.(0) in
            for k = 0 to i - 1 do
              res.(k) <- t.(k)
            done;
            for k = i to (n - 2) do
              res.(k) <- t.(k + 1)
            done;
            res)
    else t
;;
```

Définition

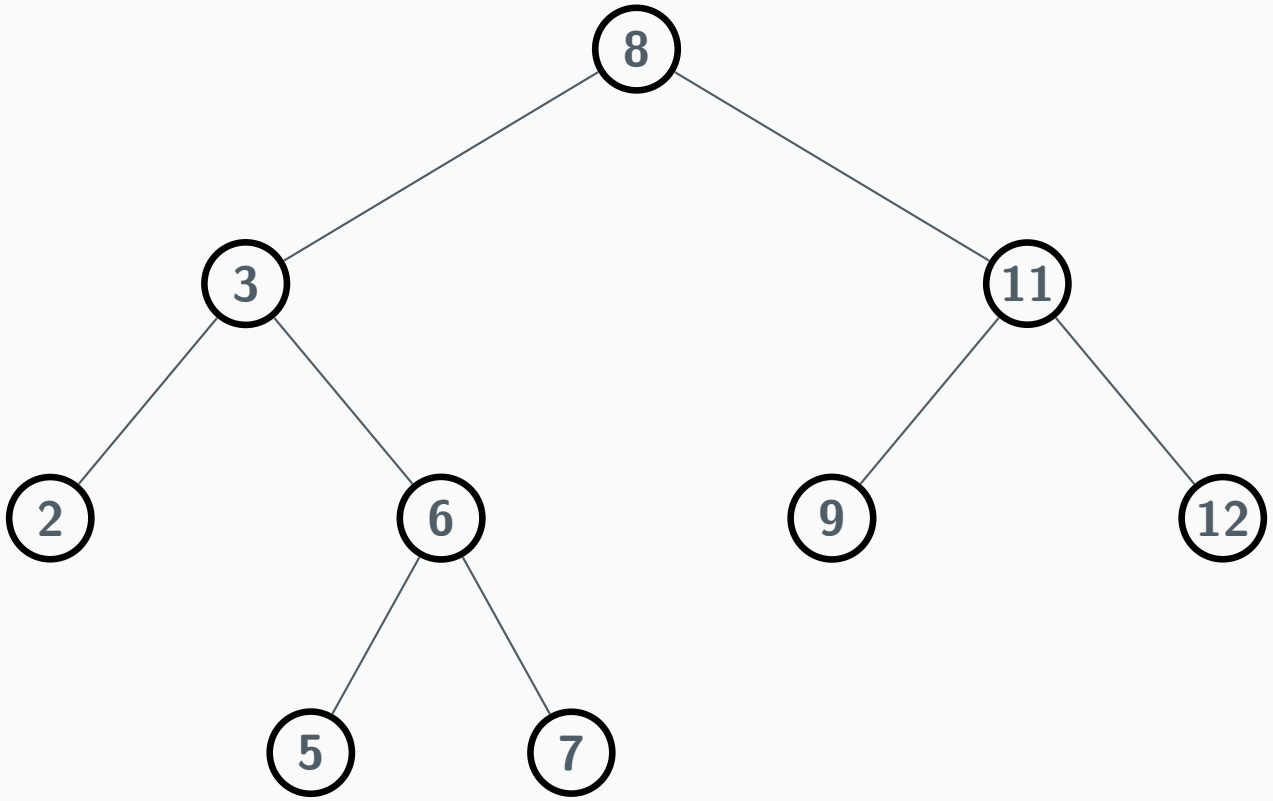
Défintion

Définition 2

*On appelle **arbre binaire de recherche** un arbre binaire étiqueté par des éléments dans un ensemble ordonné tel que l'étiquette de chaque nœud est supérieure à toutes les étiquettes des nœuds du sous arbre de gauche et inférieure à toutes les étiquettes des nœuds du sous arbre de droite.*

Exemple

L'arbre suivant est un arbre binaire de recherche.



Remarques

- Nous utiliserons souvent des entiers comme étiquettes mais on peut aussi utiliser des lettres ou des mots triés par ordre alphabétique.

Remarques

- Nous utiliserons souvent des entiers comme étiquettes mais on peut aussi utiliser des lettres ou des mots triés par ordre alphabétique.
- On peut formaliser cela. Si on note, pour un arbre a, $E(a)$ l'ensemble de ses étiquettes alors un arbre binaire est un arbre binaire de recherche s'il est vide ou de la forme $N(x, g, d)$ où,
  -

Remarques

- Nous utiliserons souvent des entiers comme étiquettes mais on peut aussi utiliser des lettres ou des mots triés par ordre alphabétique.
- On peut formaliser cela. Si on note, pour un arbre a, $E(a)$ l'ensemble de ses étiquettes alors un arbre binaire est un arbre binaire de recherche s'il est vide ou de la forme $N(x, g, d)$ où,
  - $\forall y \in E(g), y < x$
  -

Remarques

- Nous utiliserons souvent des entiers comme étiquettes mais on peut aussi utiliser des lettres ou des mots triés par ordre alphabétique.
- On peut formaliser cela. Si on note, pour un arbre a, $E(a)$ l'ensemble de ses étiquettes alors un arbre binaire est un arbre binaire de recherche s'il est vide ou de la forme $N(x, g, d)$ où,
  - $\forall y \in E(g), y < x$
  - $\forall y \in E(d), y > x$
  -

Remarques

- Nous utiliserons souvent des entiers comme étiquettes mais on peut aussi utiliser des lettres ou des mots triés par ordre alphabétique.
- On peut formaliser cela. Si on note, pour un arbre a, $E(a)$ l'ensemble de ses étiquettes alors un arbre binaire est un arbre binaire de recherche s'il est vide ou de la forme $N(x, g, d)$ où,
  - $\forall y \in E(g), y < x$
  - $\forall y \in E(d), y > x$
  - les arbres g et d sont des arbres binaires de recherche.

Remarques

- Nous utiliserons souvent des entiers comme étiquettes mais on peut aussi utiliser des lettres ou des mots triés par ordre alphabétique.
- On peut formaliser cela. Si on note, pour un arbre a, $E(a)$ l'ensemble de ses étiquettes alors un arbre binaire est un arbre binaire de recherche s'il est vide ou de la forme $N(x, g, d)$ où,
  - $\forall y \in E(g), y < x$
  - $\forall y \in E(d), y > x$
  - les arbres g et d sont des arbres binaires de recherche.

  Comme on le voit, on a donc une définition récursive, cela implique que l'on pourra faire des preuves par des raisonnements par induction.

Remarques

- Il existe trois parcours d'un arbre binaire. En effet, en parcourant l'arbre de gauche à droite et en considérant les noeuds vides, on va passer par tous les sommets trois fois. On peut donc définir trois parcours :
  - Le parcours préfixe où l'on liste le nœud la première fois où on le rencontre. Dans notre exemple :

Remarques

- Il existe trois parcours d'un arbre binaire. En effet, en parcourant l'arbre de gauche à droite et en considérant les noeuds vides, on va passer par tous les sommets trois fois. On peut donc définir trois parcours :
  - Le parcours préfixe où l'on liste le nœud la première fois où on le rencontre. Dans notre exemple :

$$8 - 3 - 2 - 6 - 5 - 7 - 11 - 9 - 12.$$

  - le parcours postfixe où l'on liste le nœud la dernière fois où on le rencontre. Dans notre exemple :

Remarques

- Il existe trois parcours d'un arbre binaire. En effet, en parcourant l'arbre de gauche à droite et en considérant les noeuds vides, on va passer par tous les sommets trois fois. On peut donc définir trois parcours :
  - Le parcours préfixe où l'on liste le nœud la première fois où on le rencontre. Dans notre exemple :

$$8 - 3 - 2 - 6 - 5 - 7 - 11 - 9 - 12.$$

  - le parcours postfixe où l'on liste le nœud la dernière fois où on le rencontre. Dans notre exemple :

$$2 - 5 - 7 - 6 - 3 - 9 - 12 - 11 - 8.$$

  - le parcours infixe où l'on liste le nœud lors de sa deuxième rencontre. Dans notre exemple :

Remarques

- Il existe trois parcours d'un arbre binaire. En effet, en parcourant l'arbre de gauche à droite et en considérant les noeuds vides, on va passer par tous les sommets trois fois. On peut donc définir trois parcours :
  - Le parcours préfixe où l'on liste le nœud la première fois où on le rencontre. Dans notre exemple :
    $$8 - 3 - 2 - 6 - 5 - 7 - 11 - 9 - 12.$$
  - le parcours postfixe où l'on liste le nœud la dernière fois où on le rencontre. Dans notre exemple :
    $$2 - 5 - 7 - 6 - 3 - 9 - 12 - 11 - 8.$$
  - le parcours infixe où l'on liste le nœud lors de sa deuxième rencontre. Dans notre exemple :
    $$2 - 3 - 5 - 6 - 7 - 8 - 9 - 11 - 12.$$

Remarque

- On peut définir un arbre binaire de recherche par le fait que lors du parcours infixe, la liste obtenue est croissante.
- En fonction de ce que l'on veut faire, on demande à ce que toutes les étiquettes soient différentes ou pas. Si on autorise à avoir des doublons, on doit faire un choix, par exemple, les étiquettes du sous-arbre de gauche sont **inférieures ou égales** à l'étiquette du noeud et les étiquettes du sous-arbre de droite sont **strictement supérieures** à l'étiquette du noeud.

Recherche

Recherche

L'utilité d'un arbre binaire de recherche est de pouvoir rechercher facilement (et rapidement) si un élément est dans l'arbre (nous verrons plus bas comment en déduire un structure de dictionnaire).

Le principe est de mettre en place une recherche semblable à la recherche dichotomique dans un tableau trié.

Le principe de l'algorithme est le suivant. Pour chercher un élément x dans un arbre binaire de recherche :

- Si l'arbre est vide (cas de base) l'élément n'est pas dans l'arbre.
- Si la racine est x : c'est gagné !
- Si x est strictement supérieur à la racine, on cherche x dans le sous-arbre de droite.
- Si x est strictement inférieur à la racine, on cherche x dans le sous-arbre de gauche.

Implémentation

Cela donne en CAML avec l'implémentation d'arbre binaire complet vu précédemment

```
type arbreBin =
  |Vide
  |N of int * arbreBin* arbreBin;;
```

Implémentation

Cela donne en CAML avec l'implémentation d'arbre binaire complet vu précédemment

```
type arbreBin =
  |Vide
  |N of int * arbreBin* arbreBin;;
```

```
let rec cherche x t = match t with
   | Vide -> false
   | N(y,g,d) when x = y -> true
   | N(y,g,d) when x > y -> cherche x d
   | N(y,g,d) when x < y -> cherche x g;;
```

Exercice

Adapter cette fonction au cas d'un `'a arbre Bin`défini par

```
type 'a arbreBin =
  |Vide
  |N of 'a*'a arbreBin*'a arbreBin;;
```

où on suppose avoir une fonction `comparaison : 'a -> 'a -> int`tel que `comparaison x y`vaut 0 si $x = y$, est strictement positif si x est plus grand que y et strictement négatif sinon.

Complexité de la recherche

Théorème 1 (Complexité de la recherche dans un arbre binaire de recherche)

L'algorithme précédent permet de trouver (ou pas) un élément dans un arbre binaire en un temps $O(h)$ où h est la hauteur de l'arbre.

Insertion

Insertion

On dispose d'un arbre binaire de recherche et d'un élément x que l'on veut insérer dans l'arbre. On veut bien évidemment garder un arbre binaire de recherche. Le principe est de faire comme si on cherchait x.

- Si on le trouve alors on ne fait rien (même si on pourrait insérer une deuxième copie)
- Si on ne le trouve pas, c'est donc que l'on est à une feuille ou à un arbre vide, on insère l'élément.

Implémentation

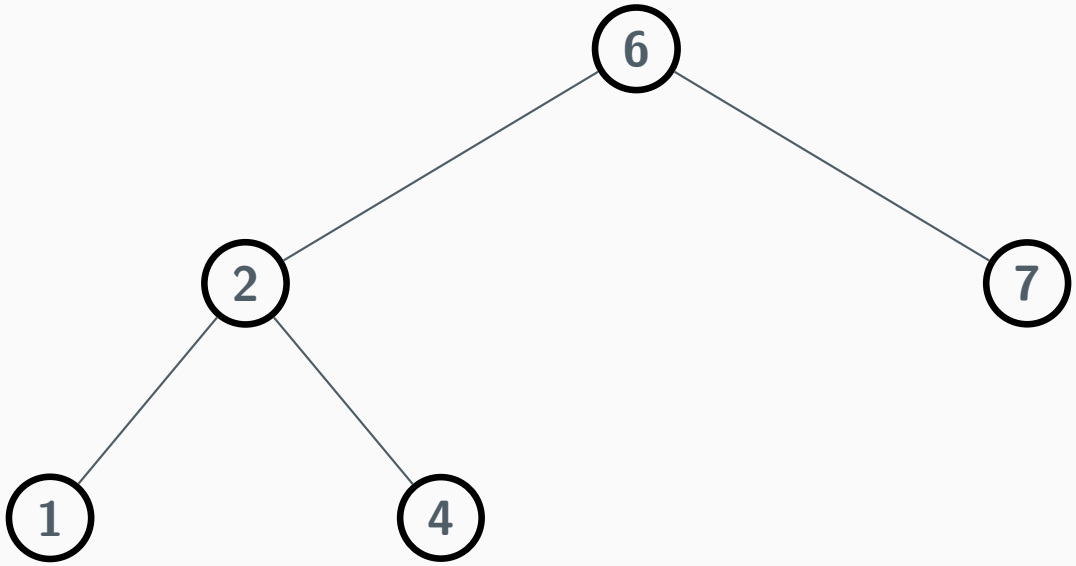
```
let rec insert x t = match t with
    | Vide -> N(x,Vide,Vide)
    | N(y,g,d) when x = y -> N(y,g,d)
    | N(y,g,d) when x < y -> N(y, insert x g,d)
    | N(y,g,d) when x > y -> N(y, g, insert x d);;
```

Exercice

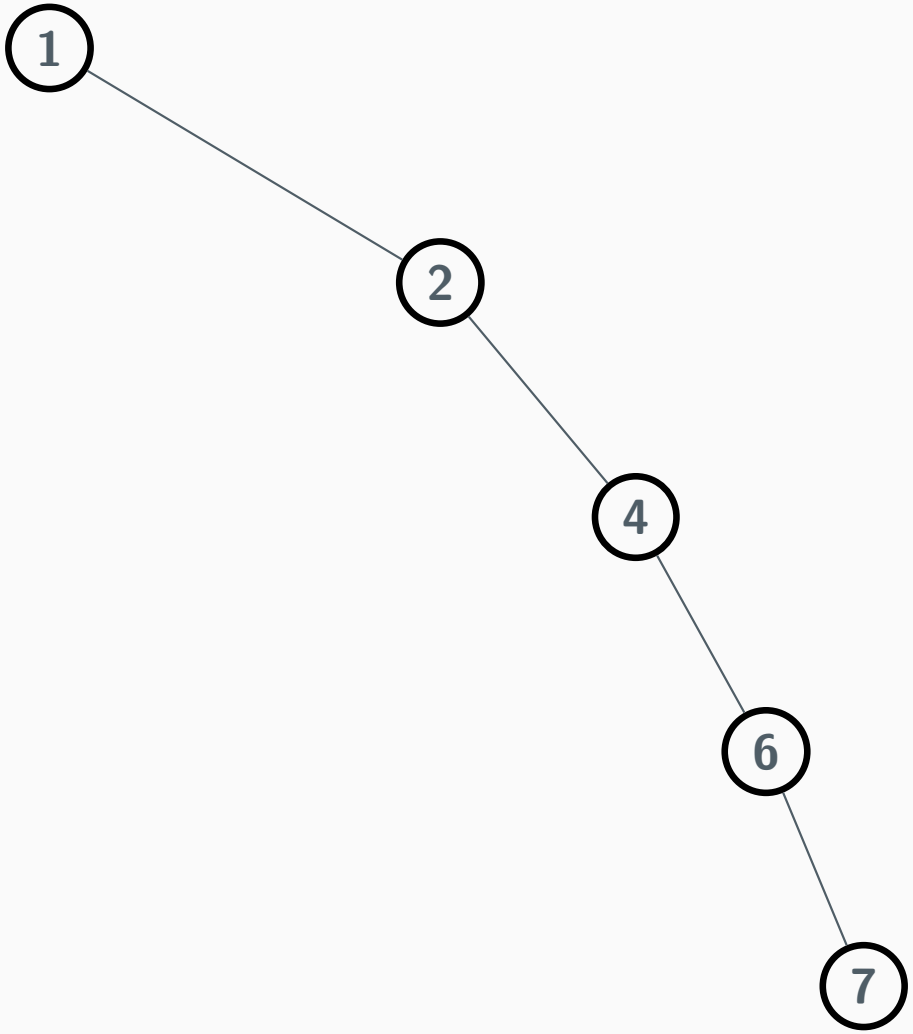
Quels arbres obtient-on en appliquant cet algorithme aux listes `[1;2;4;6;7]` et `[6;2;4;1;7]` ?

Exercice

Quels arbres obtient-on en appliquant cet algorithme aux listes `[1;2;4;6;7]` et `[6;2;4;1;7]` ?



Exercice



Exercice

Écrire une fonction de type `int list -> arbre` qui insère successivement les termes d'une liste dans un arbre binaire de recherche.

Exercice

Écrire une fonction de type `int list -> arbre` qui insère successivement les termes d'une liste dans un arbre binaire de recherche.

```
let  insertListe  l =
^^Ilet rec aux lis a = match lis with
^^I^^I|[] -> a
^^I^^I| t::q -> aux q (insert t a)
^^Iin aux l Vide
^^I;;
```

Complexité de l'insertion

Théorème 2 (Complexité de l'insertion dans un arbre binaire de recherche)

L'algorithme précédent permet d'insérer un élément dans un arbre binaire en un temps $O(h)$ où h est la hauteur de l'arbre.

IMPORTANT

Attention

On a vu que la recherche et l'insertion ont une complexité en $O(h)$. Or, pour une taille d'arbre donné (disons n), la hauteur de l'arbre peut varier de n (dans le cas d'un arbre « peigne ») à $\log(n)$ dans le cas d'un arbre complet.
C'est pour cela que l'on va essayer de garder un arbre le plus « étoffé »possible où on peut espérer avoir $h \simeq \log(n)$.

Attention

On peut ainsi construire un arbre binaire de recherche à partir d'une liste en insérant les uns après les autres les éléments dans un arbre initialement vide. Il faut noter que si la liste est initialement triée, on obtient un arbre « peigne ».
Il existe des algorithmes qui permettent de construire un ABR de sorte que sa profondeur soit $O(\log(n))$. On peut donc construire un tel arbre en

$$\sum_{k=1}^{n} \ln(k) \sim$$

Attention

On peut ainsi construire un arbre binaire de recherche à partir d'une liste en insérant les uns après les autres les éléments dans un arbre initialement vide. Il faut noter que si la liste est initialement triée, on obtient un arbre « peigne ».
Il existe des algorithmes qui permettent de construire un ABR de sorte que sa profondeur soit $O(\log(n))$. On peut donc construire un tel arbre en

$$\sum_{k=1}^{n} \ln(k) \sim n \ln(n)$$

Suppression

Suppression

L'algorithme de suppression est un peu plus compliqué. Bien évidemment, supprimer une feuille est simple, par contre, si c'est un noeud interne il faut « boucher le trou ».

On voit que l'on peut, par récursivité, se ramener au cas où l'on supprime la racine.

Suppression

L'algorithme de suppression est un peu plus compliqué. Bien évidemment, supprimer une feuille est simple, par contre, si c'est un noeud interne il faut « boucher le trou ».

On voit que l'on peut, par récursivité, se ramener au cas où l'on supprime la racine.

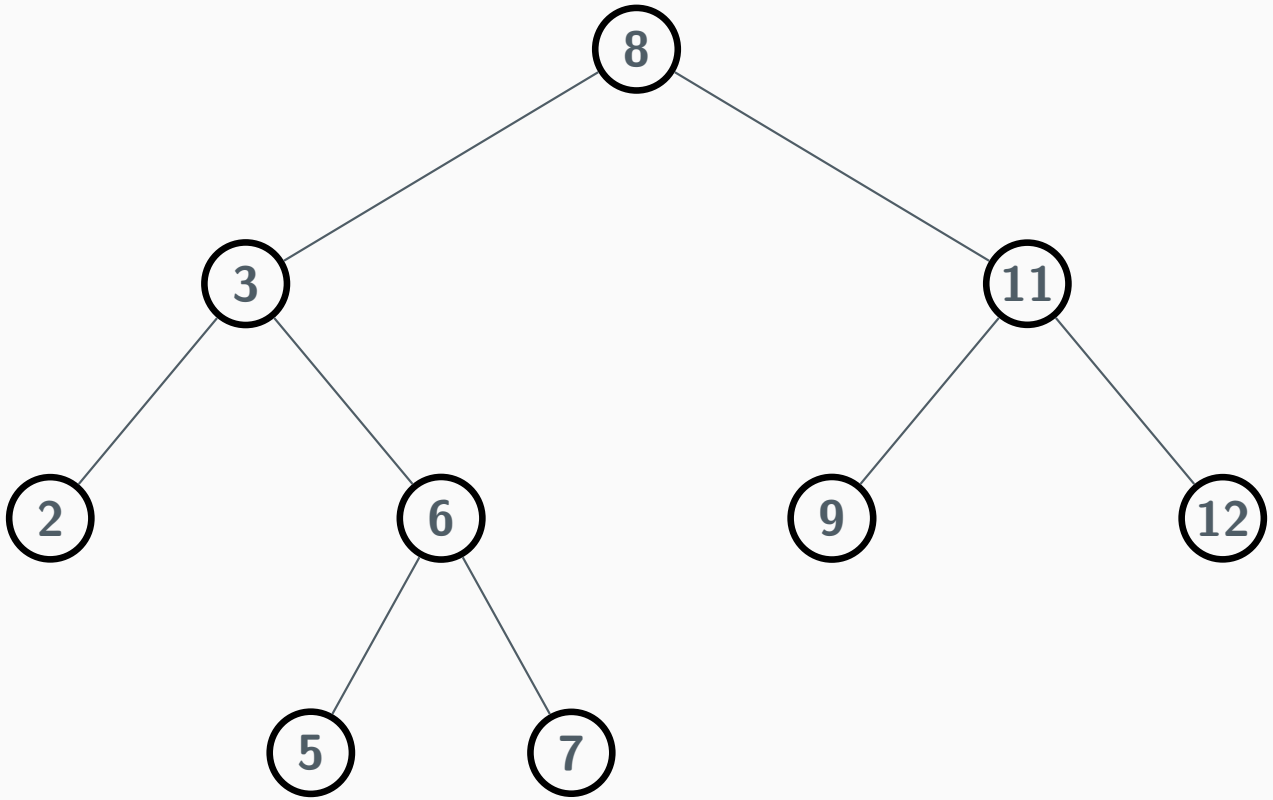
- Si le sous-arbre de gauche est vide, on remplace l'arbre par son sous-arbre de droite
- Si le sous-arbre de gauche n'est pas vide :
  - on cherche le plus grand élément du sous-arbre de gauche (en allant toujours à droite)
  - on enlève cet élément du sous-arbre de gauche en remarquant que son fils droit est nécessairement l'arbre vide
  - on place cet élément au sommet du sous-arbre de gauche

Remarque

Cette méthode permet d'assurer que la hauteur de l'arbre ne peut que descendre.

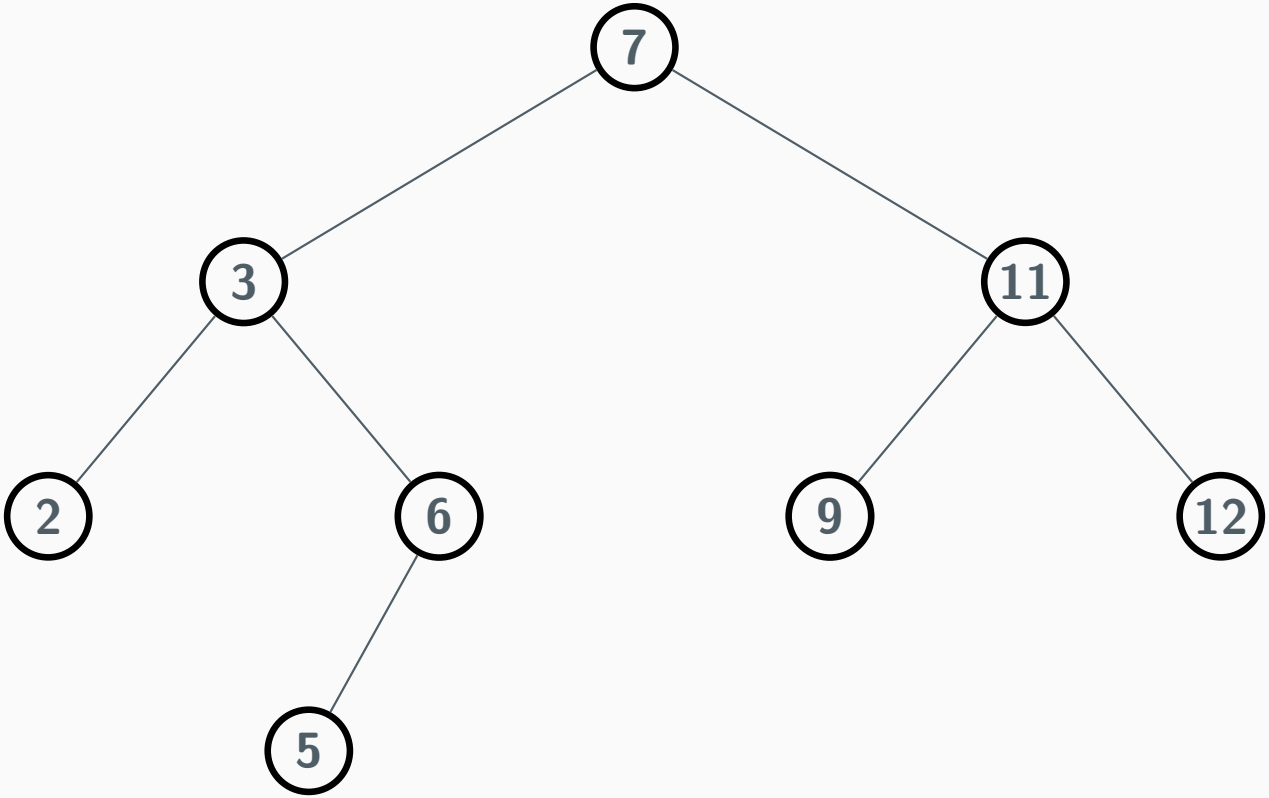
Exemple

Si on considère l'arbre binaire de recherche suivant et que l'on veut enlever la racine



Exemple

On obtient

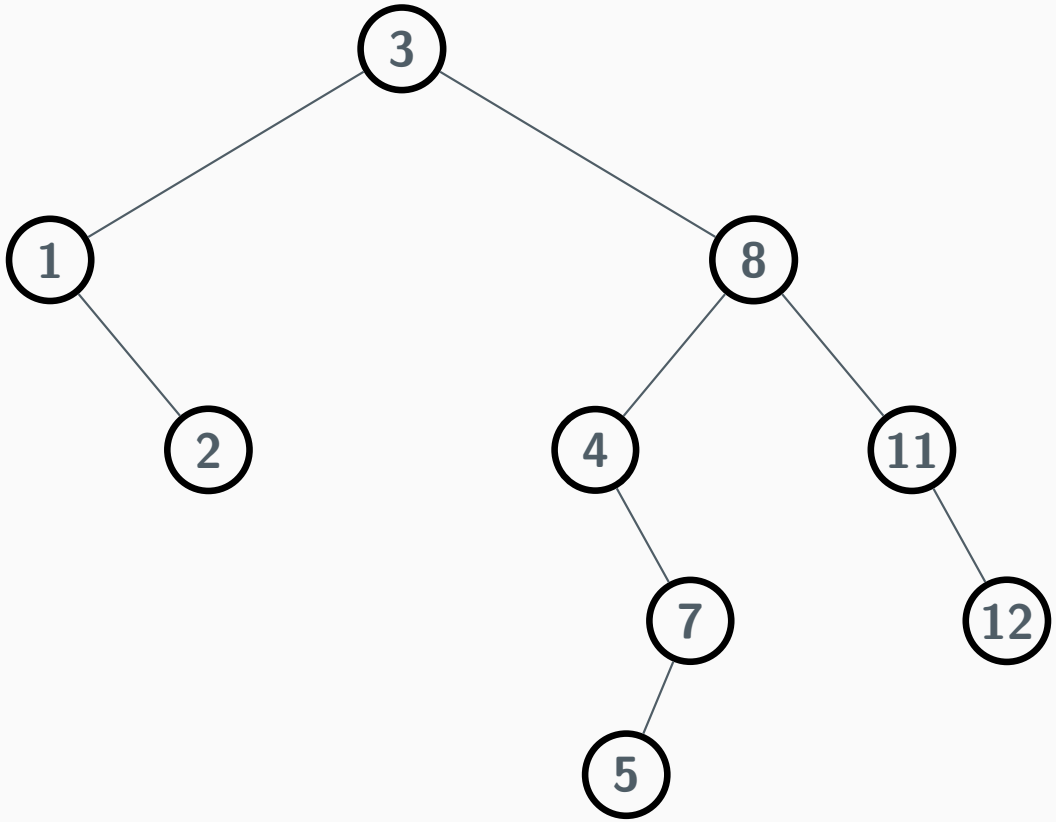


Exercice

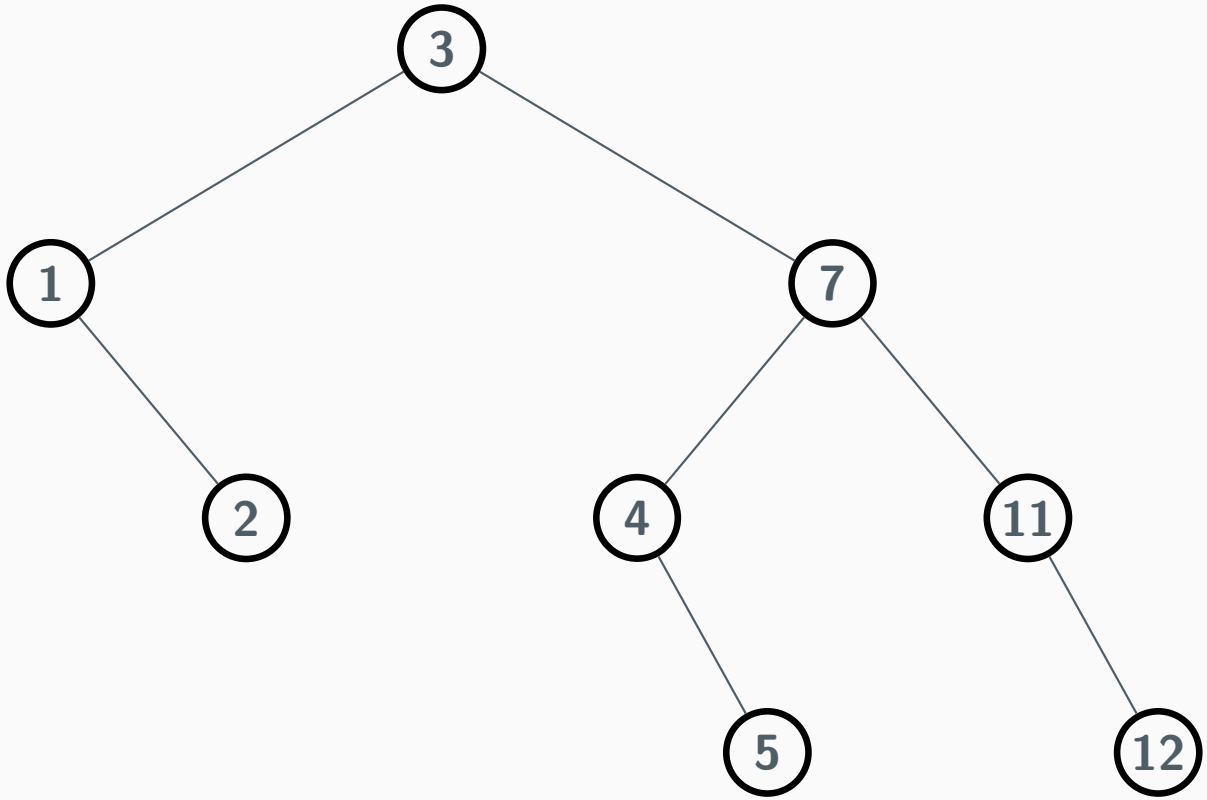
On considère l'arbre binaire de recherche en insérant successivement les éléments de la liste `[3;1;8;11;4;7;12;2;5]`. Dessiner l'arbre obtenu puis dessiner l'arbre obtenu en supprimant 8.

Exercice

On considère l'arbre binaire de recherche en insérant successivement les éléments de la liste `[3;1;8;11;4;7;12;2;5]`. Dessiner l'arbre obtenu puis dessiner l'arbre obtenu en supprimant 8.



Exercice



Suppression

On en déduit la fonction :

```
let supprime_racine a =
  match a with
    | Vide -> failwith "impossible"
    | N(Vide,_,d) -> d
    | N(g,x,d) -> let rec recMax a =
     match a with
      |Vide -> failwith "impossible"
      |N(g,x,Vide) -> (x,g)
      |N(g,x,d) -> let (v,ab) = recMax d in (v,N(g,x,ab))
in let (v,ab) = recMax g in N(ab,v,d);;
```

où `recMax : arbreBin -> int*arbreBin` associe à un arbre, le couple dont le premier élément est le maximum et le deuxième est l'arbre obtenu en enlevant ce maximum.

Suppression

Et donc le programme

```
let rec supprime a x =
match a with
| Vide -> Vide
| N(g,y,d) when x > y -> N(g,y,supprime d x)
| N(g,y,d) when x < y -> N(supprime g x , y , d)
| N(g,y,d) when x = y -> supprime_racine a;;
```

Complexité de la suppression

Théorème 3 (Complexité de la suppression dans un arbre binaire de recherche)

L'algorithme précédent permet de supprimer un élément dans un arbre binaire en un temps $O(h)$ où h est la hauteur de l'arbre.

Suppression

Dictionnaire

Un dictionnaire est une structure de données contenant des élément repérés par une clé. L'exemple typique est le dictionnaire usuel où les éléments sont les définitions qui sont repérées par les mots qu'elles définissent. Dans un dictionnaire on veut :

- Créer un dictionnaire vide
- Insérer un élément
- Rechercher des éléments par la clé pour éventuellement le modifier
- Supprimer des éléments

Dictionnaire

On voit donc que l'on peut réaliser un dictionnaire où la recherche sera efficace en utilisant un arbre binaire de recherche. En effet, toutes les opérations (insertion, suppression, recherche) se font de manière au plus linéaire en la **hauteur** de l'arbre binaire de recherche. Si on arrive à maintenir l'arbre le plus étoffé possible (voir exercices) on peut donc garantir une complexité en $O(\ln n)$ où n est la taille du dictionnaire.

Tas et liste de priorité