Mise en œuvre des ensembles (II)

Pratique de la programmation orientée-objet

Michel Schinz – 2014–03–31

Mises en œuvre

Pour mémoire, nous avons déjà examiné deux mises en œuvre des ensembles :

1. les « listes-ensembles » (classe `ListSet`),
2. les tables de hachage (classe `HashSet`),

Nous allons maintenant examiner la dernière :

3. les arbres binaires de recherche (classe `TreeSet`).

Interface Set (rappel)

L'interface `Set` implémentée par nos mises en œuvre est une simplification de celle de la bibliothèque Java.

```
public interface Set<E> extends Iterable<E>{
  boolean isEmpty();
  int size();
  void add(E elem);
  void remove(E elem);
  boolean contains(E elem);
  Iterator<E> iterator();
}
```

Mise en œuvre n°3 : arbres de recherche

Tableaux-ensembles

Vous connaissez déjà une technique pour effectuer une recherche rapide dans un tableau d'éléments trié : la recherche dichotomique.

Pourquoi ne pas stocker les éléments de l'ensemble dans un tableau et utiliser ensuite la recherche dichotomique pour les trouver ?

Le test d'appartenance devient rapide – en O(log n) – mais malheureusement l'insertion et la suppression restent en O(n) en raison du déplacement d'éléments qu'elles impliquent.

Arbres de recherche

Pour que les opérations d'ajout et de suppression aient une complexité meilleure que O(n), il faut abandonner l'idée de stocker les éléments dans un tableau.
Toutefois, utiliser une simple liste chaînée n'est pas assez efficace, comme nous l'avons vu avec les listes-ensembles.
Il faudrait trouver une manière de stocker les éléments dans des nœuds liés entre eux – afin de pouvoir faire des ajouts et suppression rapides à des endroits quelconques – tout en gardant la possibilité de faire la recherche par dichotomie.
C'est l'idée des **arbres binaires de recherche**.

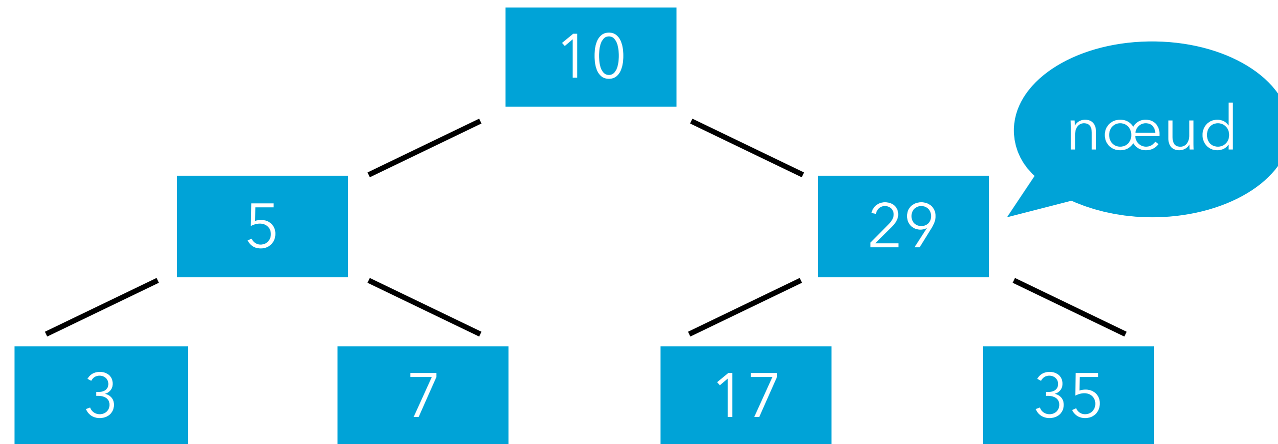
Arbre de recherche

Un **arbre (binaire) de recherche** (*binary search tree*) est un arbre composé de nœuds ayant chacun deux fils, qui peuvent être vides.
A chaque nœud est associé un élément et l'arbre est organisé de manière à ce que l'invariant suivant soit respecté :

- tous les éléments du sous-arbre gauche d'un nœud sont strictement plus petits que celui du nœud, et
- tous les éléments du sous-arbre droite d'un nœud sont strictement plus grands que celui du nœud.

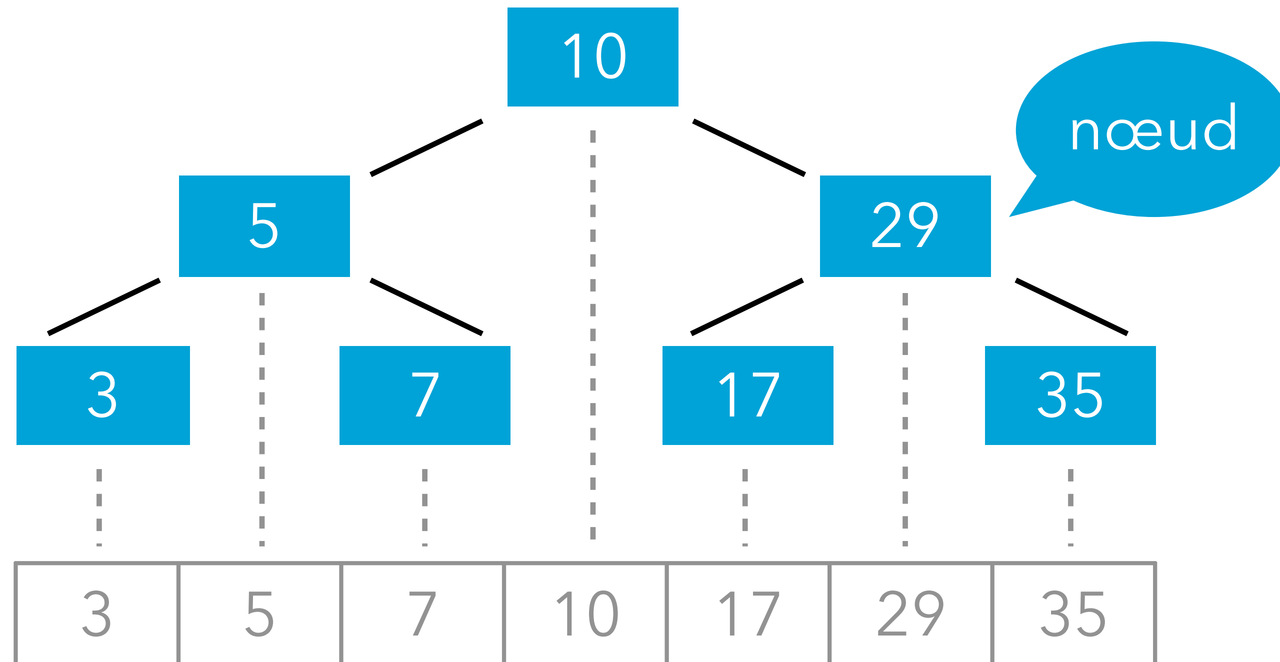
Arbre de recherche

Invariant (rappel) : pour tout nœud, tous les éléments du sous-arbre gauche sont strictement plus petits que celui du nœud, ceux du sous-arbre droite strictement plus grands.



Arbre de recherche

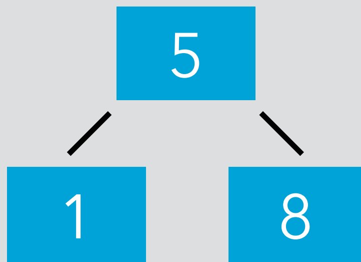
Invariant (rappel) : pour tout nœud, tous les éléments du sous-arbre gauche sont strictement plus petits que celui du nœud, ceux du sous-arbre droite strictement plus grands.



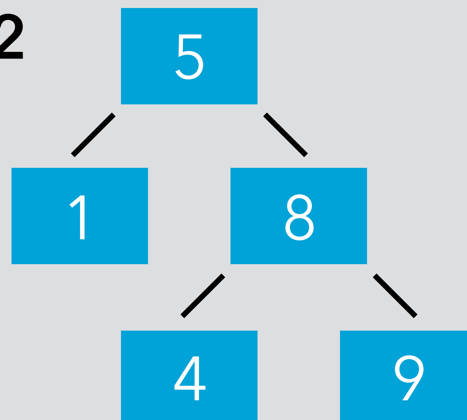
Exercice : est-ce un a.b.r. ?

Invariant (rappel) : pour tout nœud, tous les éléments du sous-arbre gauche sont strictement plus petits que celui du nœud, ceux du sous-arbre droite strictement plus grands.

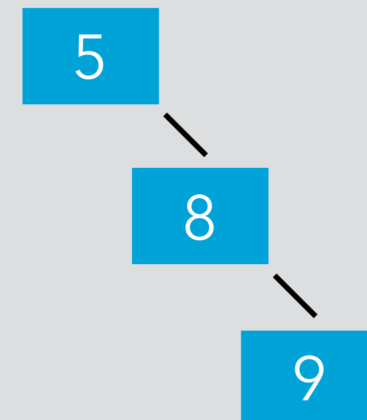
1



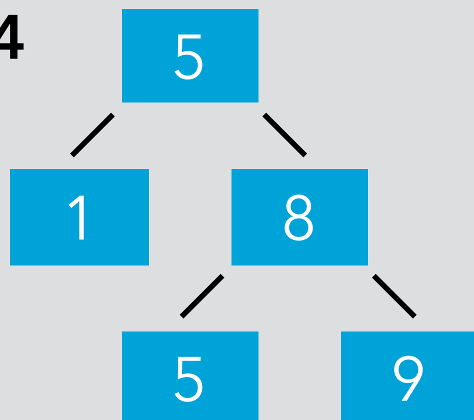
2



3

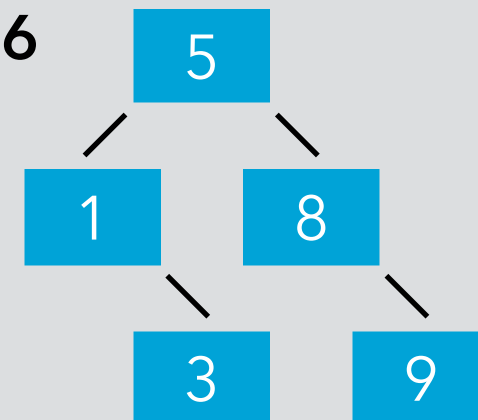


4



5

6



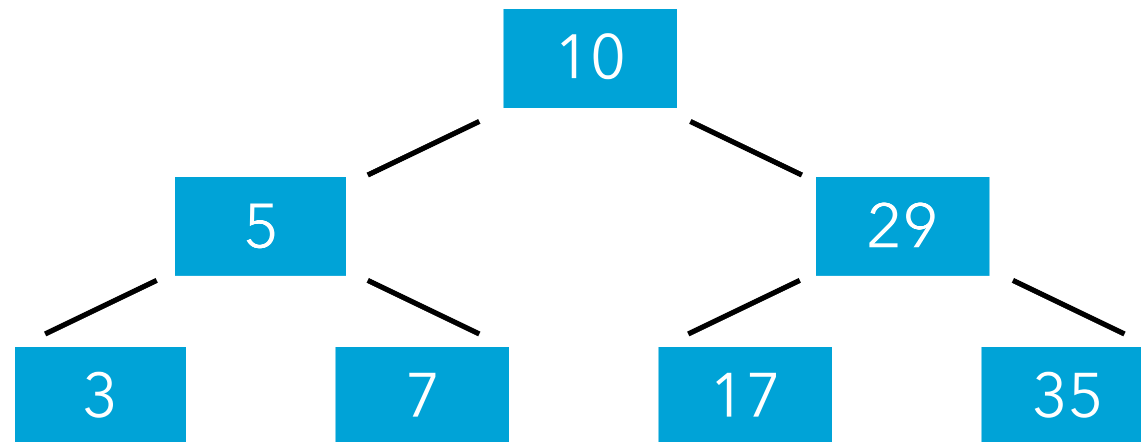
Recherche

La recherche d'un élément dans un arbre binaire de recherche est très similaire à la recherche dichotomique :

- l'élément recherché est comparé avec l'élément à la racine (correspondant à l'élément au centre du tableau),
- si les éléments sont égaux, la recherche se termine avec succès,
- si l'élément recherché est plus petit que celui à la racine, la recherche se poursuit dans le sous-arbre gauche (correspondant à la moitié inférieure du tableau),
- si l'élément recherché est plus grand que celui à la racine, la recherche se poursuit dans le sous-arbre droit (correspondant à la moitié supérieure du tableau),
- si l'arbre est vide, la recherche échoue.

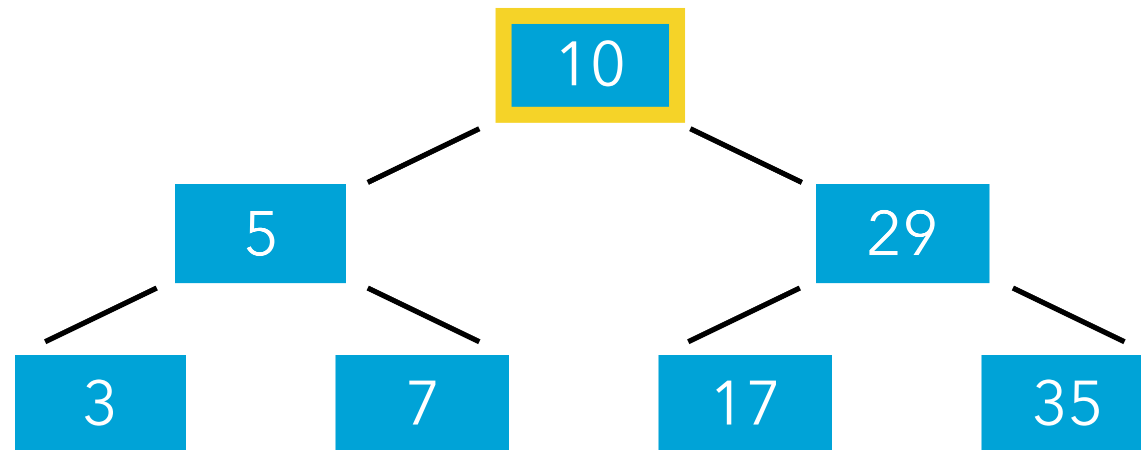
Exemple de recherche

Exemple : on recherche l'élément 17 (présent).



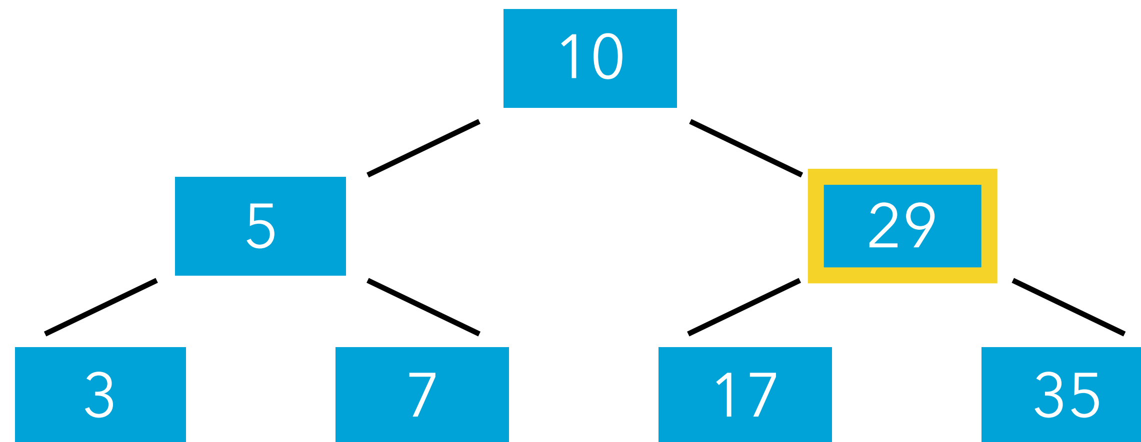
Exemple de recherche

Exemple : on recherche l'élément 17 (présent).



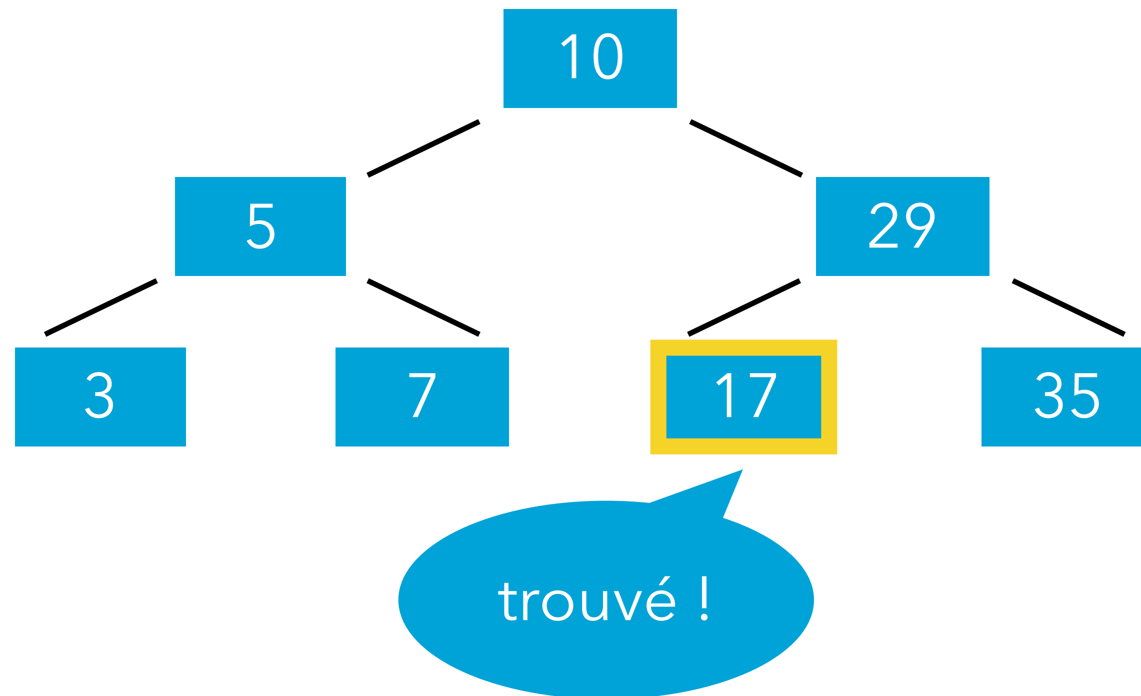
Exemple de recherche

Exemple : on recherche l'élément 17 (présent).



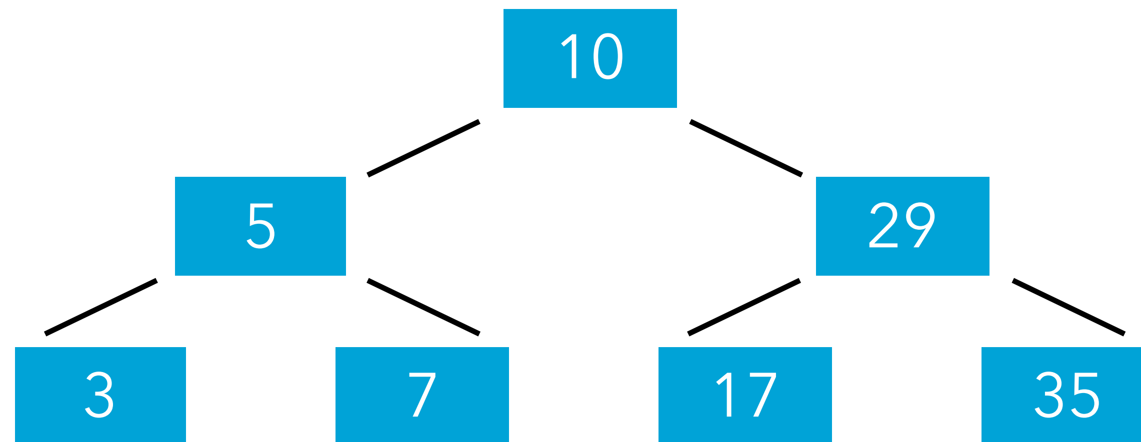
Exemple de recherche

Exemple : on recherche l'élément 17 (présent).



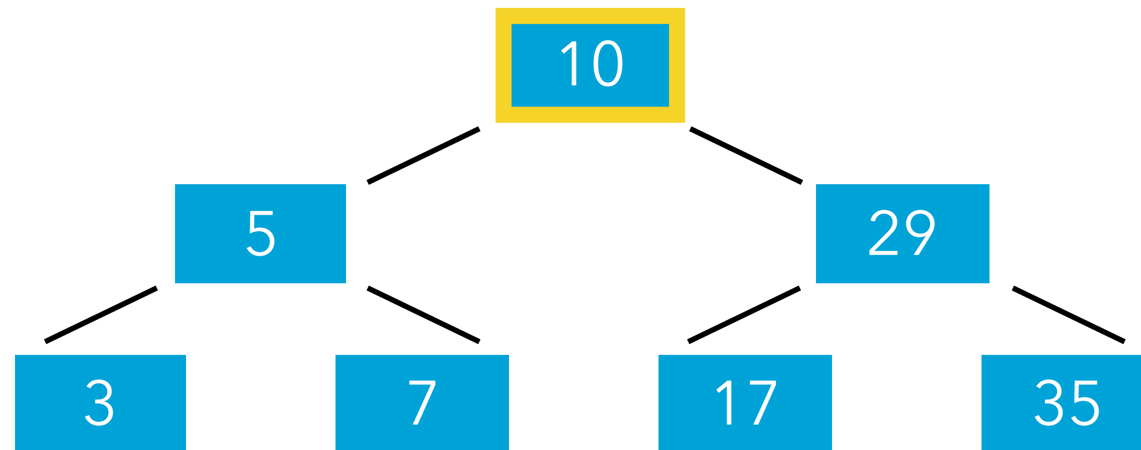
Exemple de recherche

Exemple : on recherche l'élément 8 (absent).



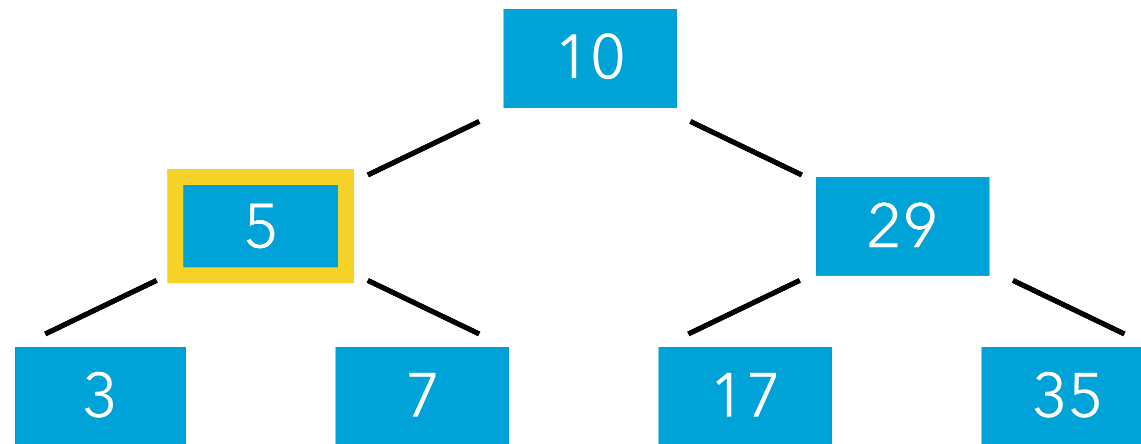
Exemple de recherche

Exemple : on recherche l'élément 8 (absent).



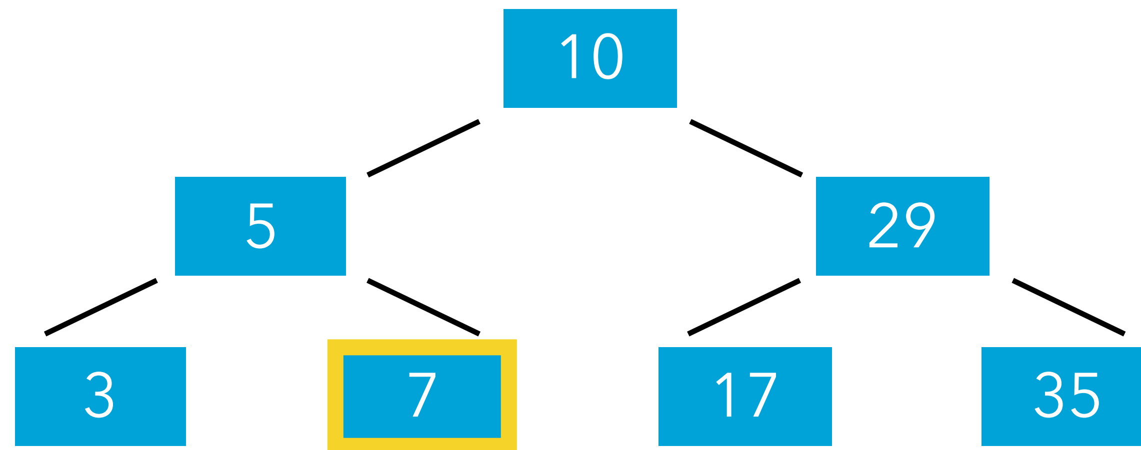
Exemple de recherche

Exemple : on recherche l'élément 8 (absent).



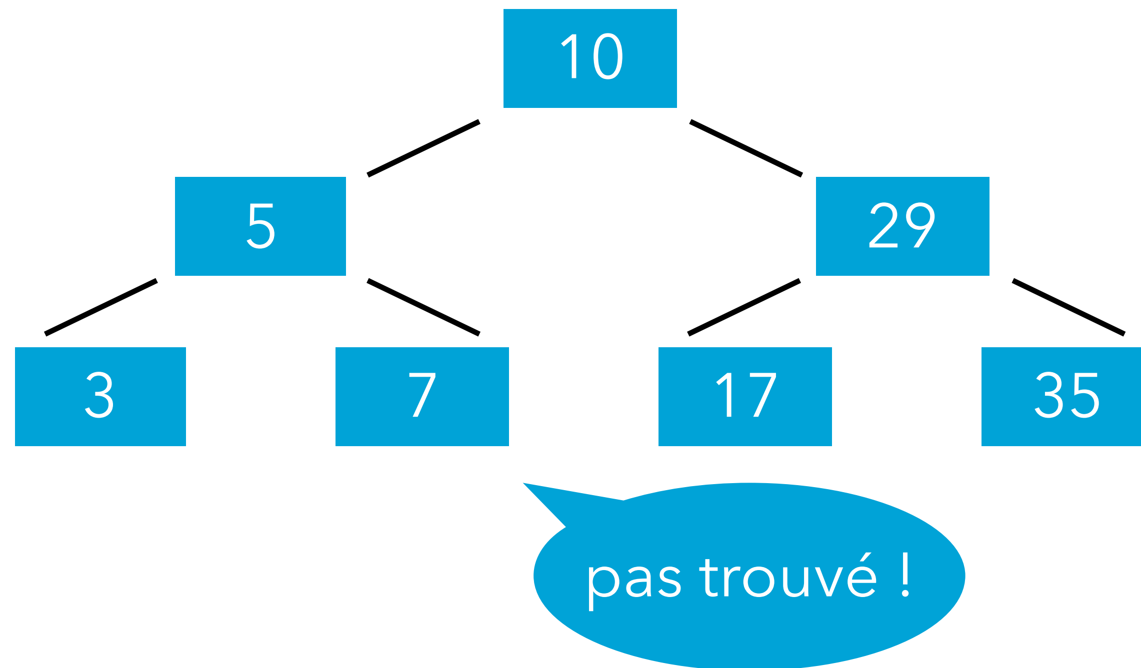
Exemple de recherche

Exemple : on recherche l'élément 8 (absent).



Exemple de recherche

Exemple : on recherche l'élément 8 (absent).



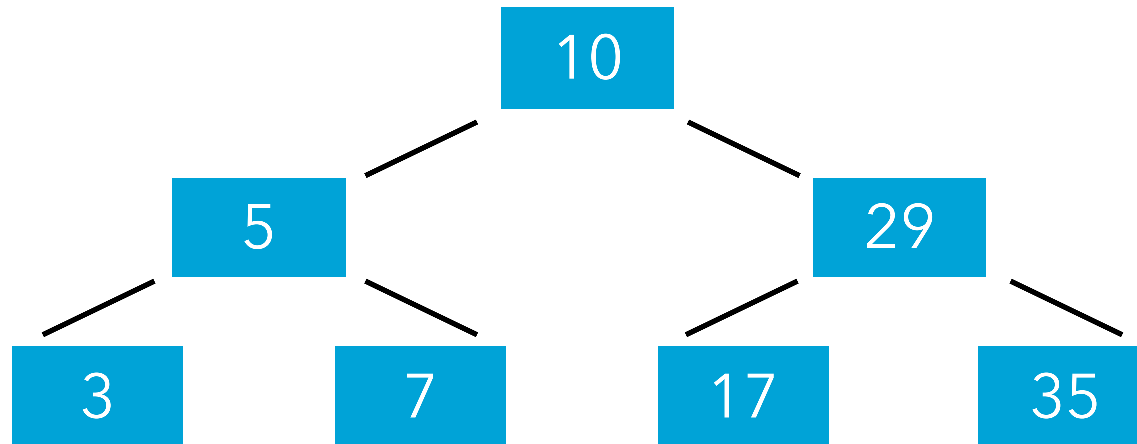
Ajout

L'ajout d'un élément à un arbre binaire de recherche ressemble beaucoup à la recherche :

- si la recherche se termine avec succès, alors l'élément est déjà présent dans l'arbre et il n'y a rien à faire,
- si la recherche échoue, alors l'élément n'est pas encore présent dans l'arbre et il faut l'insérer comme fils du dernier nœud visité par la recherche.

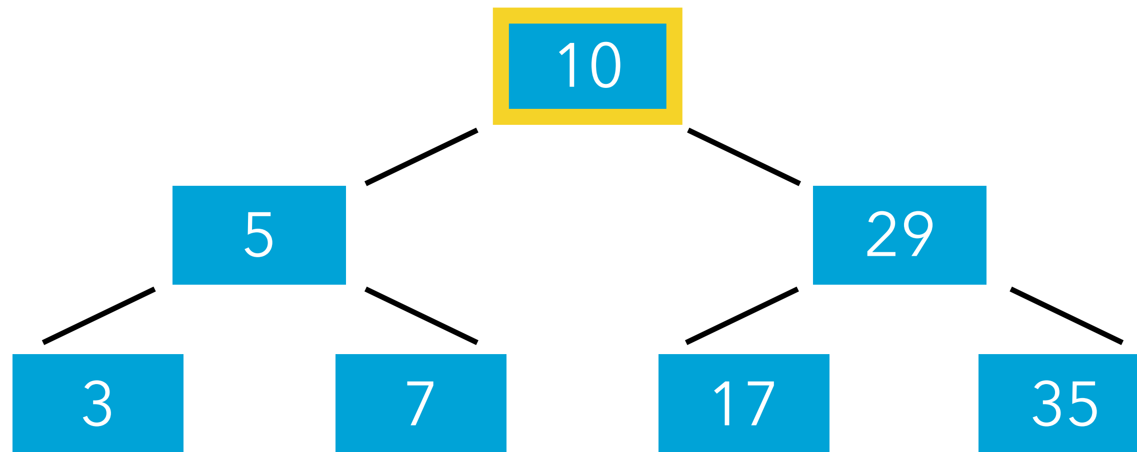
Exemple d'ajout

Exemple : on ajoute l'élément 42.



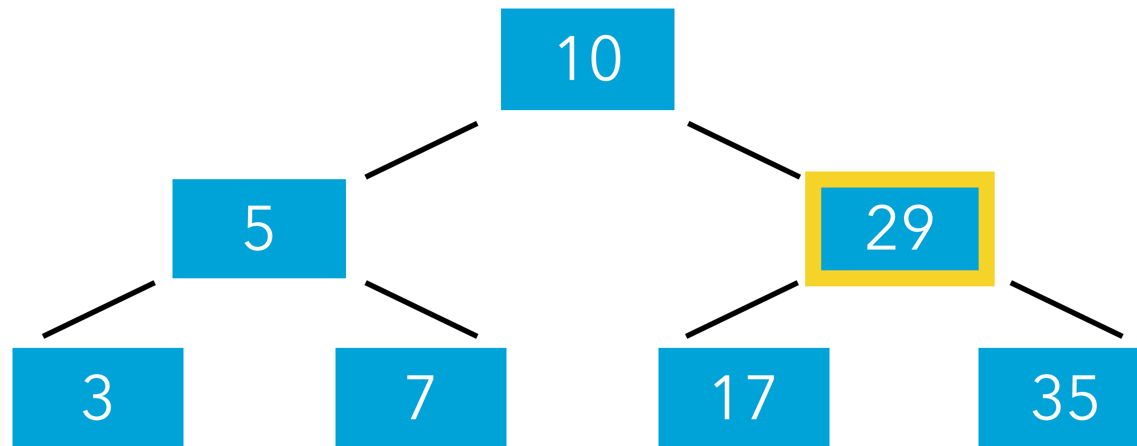
Exemple d'ajout

Exemple : on ajoute l'élément 42.



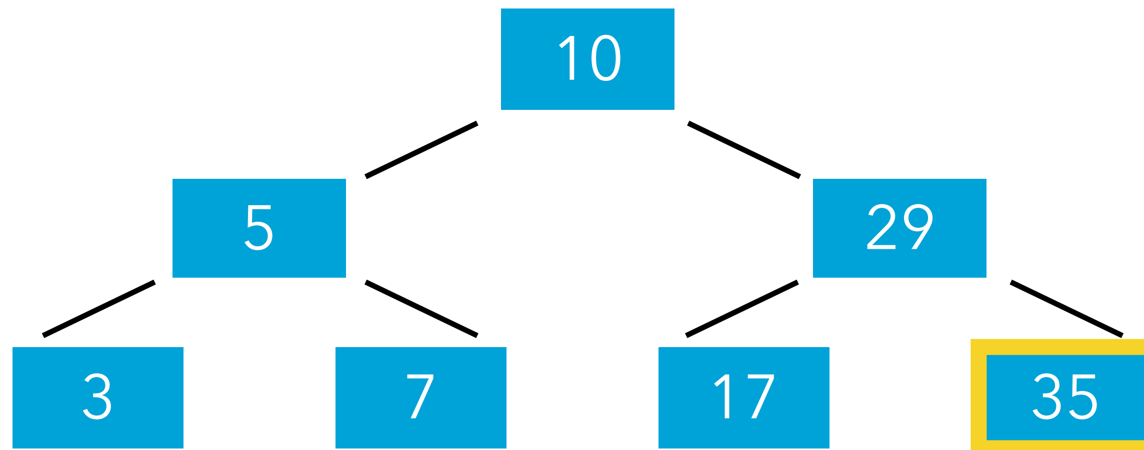
Exemple d'ajout

Exemple : on ajoute l'élément 42.



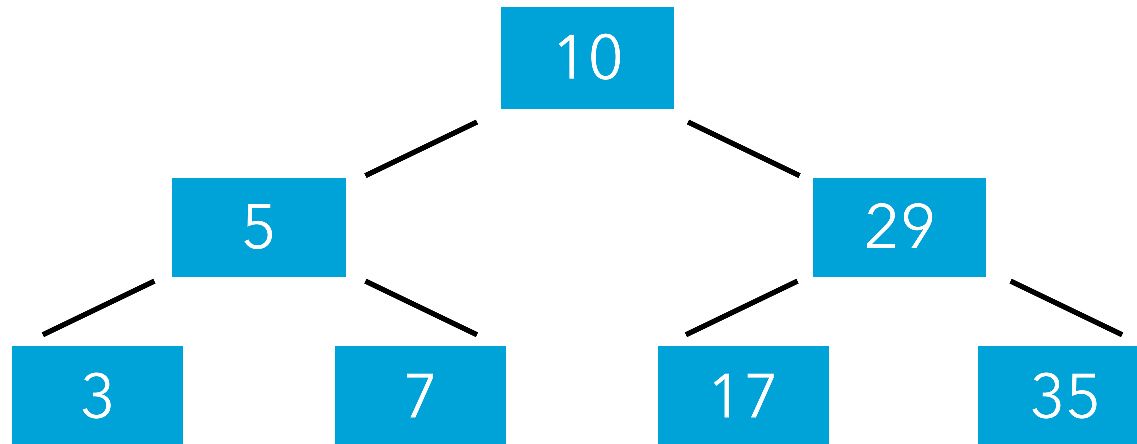
Exemple d'ajout

Exemple : on ajoute l'élément 42.



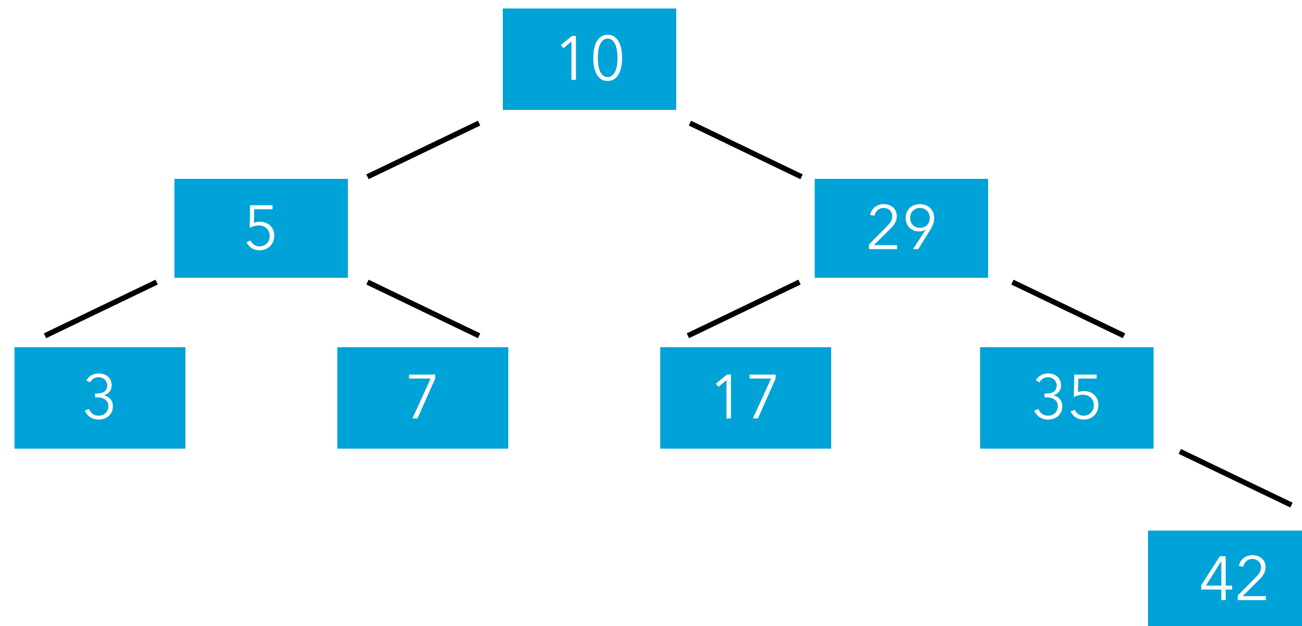
Exemple d'ajout

Exemple : on ajoute l'élément 42.



Exemple d'ajout

Exemple : on ajoute l'élément 42.



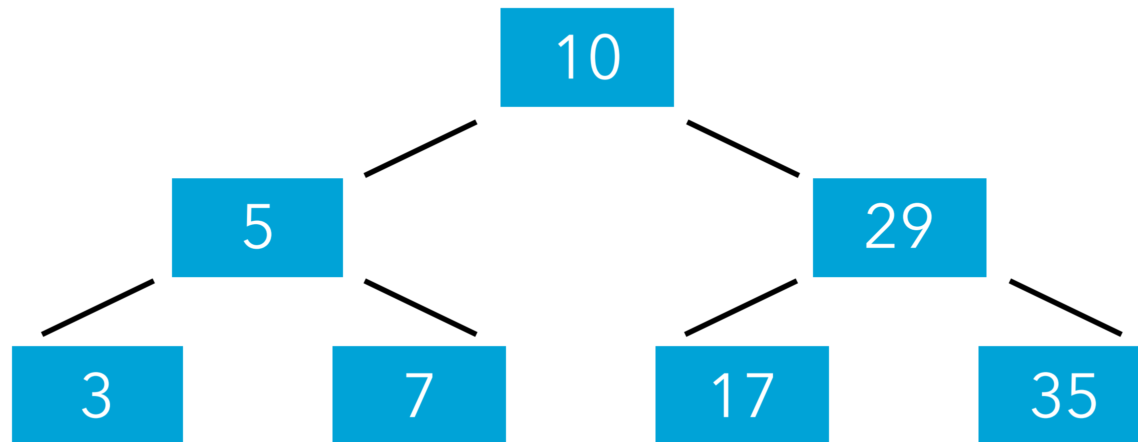
Suppression

La suppression d'un élément d'un arbre binaire de recherche commence par rechercher le nœud contenant l'élément à supprimer puis distingue trois cas :

1. si le nœud n'a aucun fils, il peut être directement supprimé,
2. si le nœud a un seul fils, il peut être remplacé par ce fils (cas similaire à une liste chaînée),
3. si le nœud a deux fils, on peut se ramener à l'un des deux premiers cas en remplaçant son élément par celui de son successeur, c-à-d le nœud contenant le prochain élément dans l'ordre.

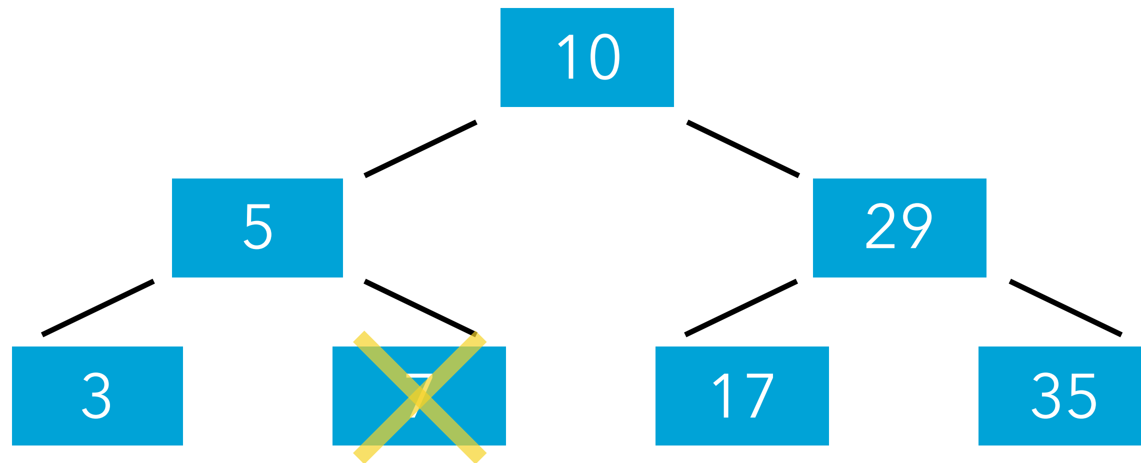
Exemple de suppression

Exemple des deux cas simples : on supprime d'abord l'élément 7, qui n'a pas de fils, puis le 5, qui n'en a qu'un.



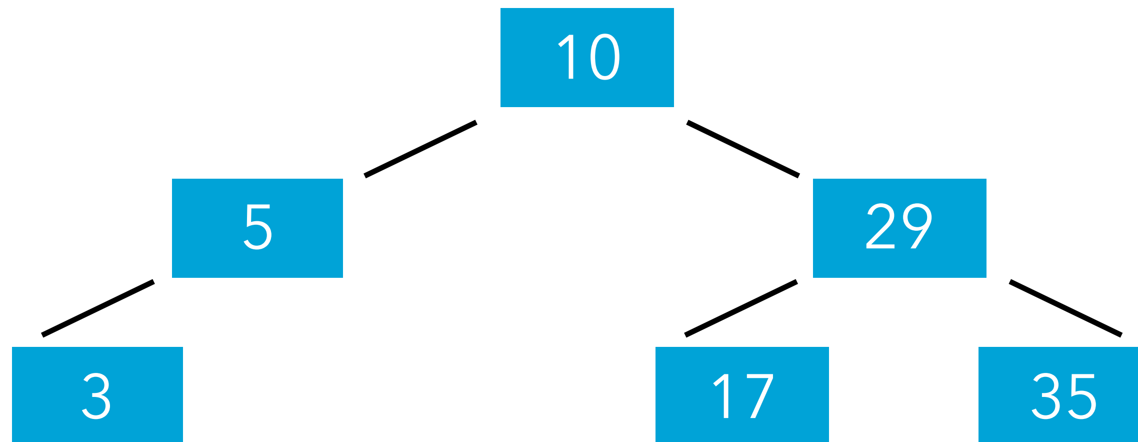
Exemple de suppression

Exemple des deux cas simples : on supprime d'abord l'élément 7, qui n'a pas de fils, puis le 5, qui n'en a qu'un.



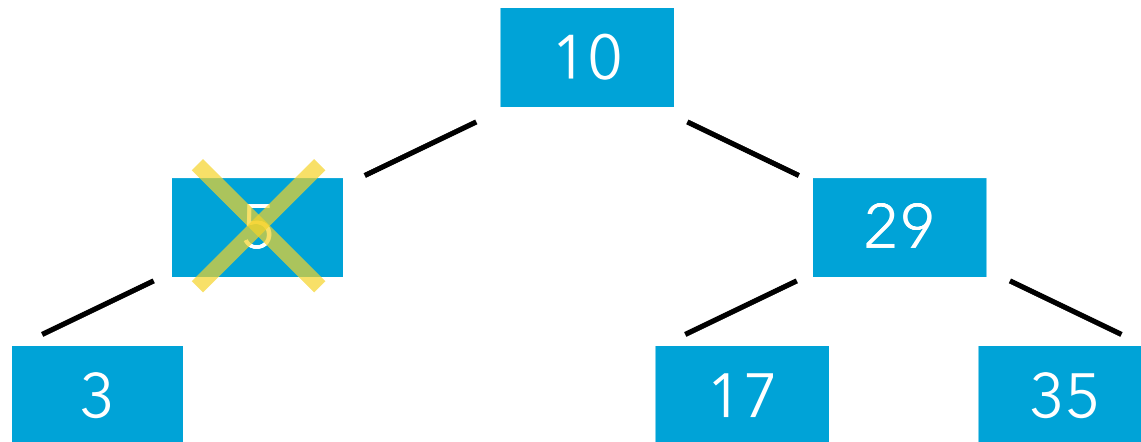
Exemple de suppression

Exemple des deux cas simples : on supprime d'abord l'élément 7, qui n'a pas de fils, puis le 5, qui n'en a qu'un.



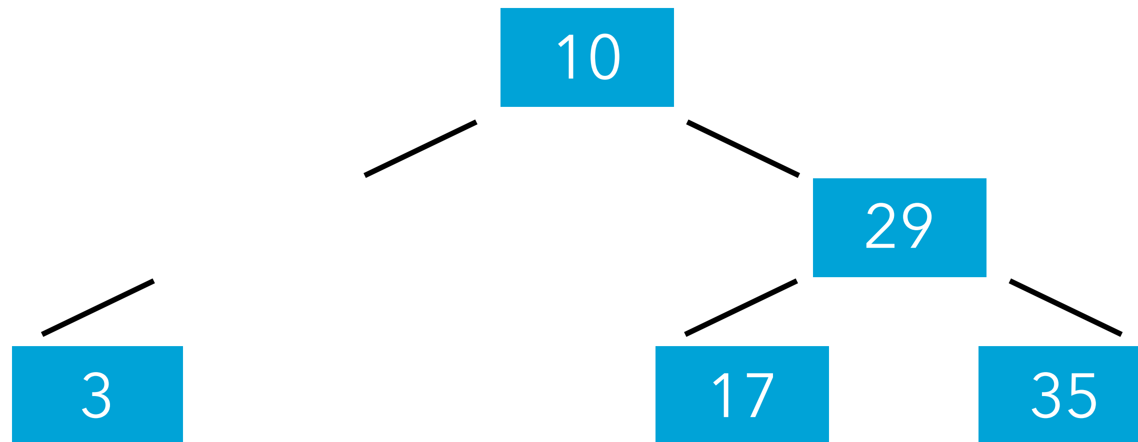
Exemple de suppression

Exemple des deux cas simples : on supprime d'abord l'élément 7, qui n'a pas de fils, puis le 5, qui n'en a qu'un.



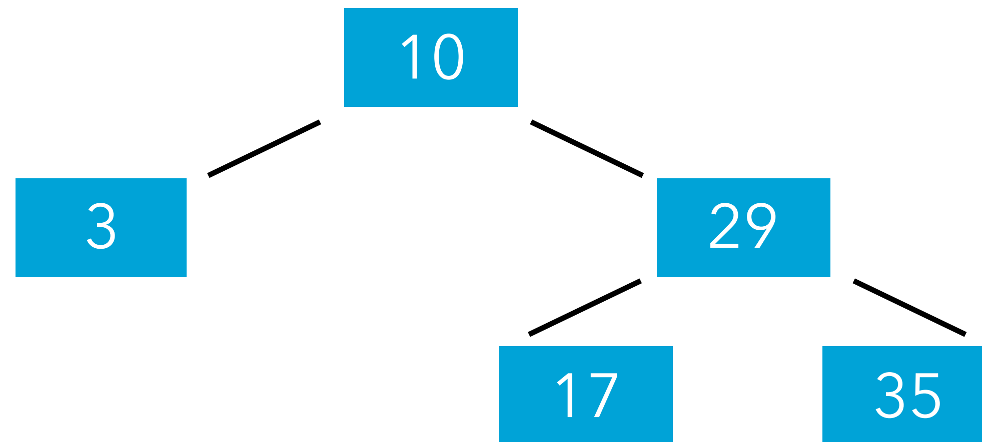
Exemple de suppression

Exemple des deux cas simples : on supprime d'abord l'élément 7, qui n'a pas de fils, puis le 5, qui n'en a qu'un.



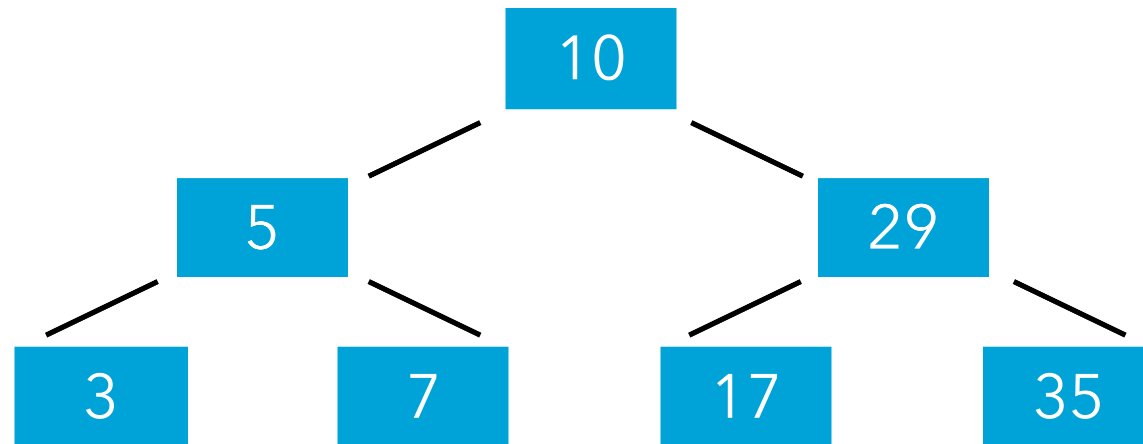
Exemple de suppression

Exemple des deux cas simples : on supprime d'abord l'élément 7, qui n'a pas de fils, puis le 5, qui n'en a qu'un.



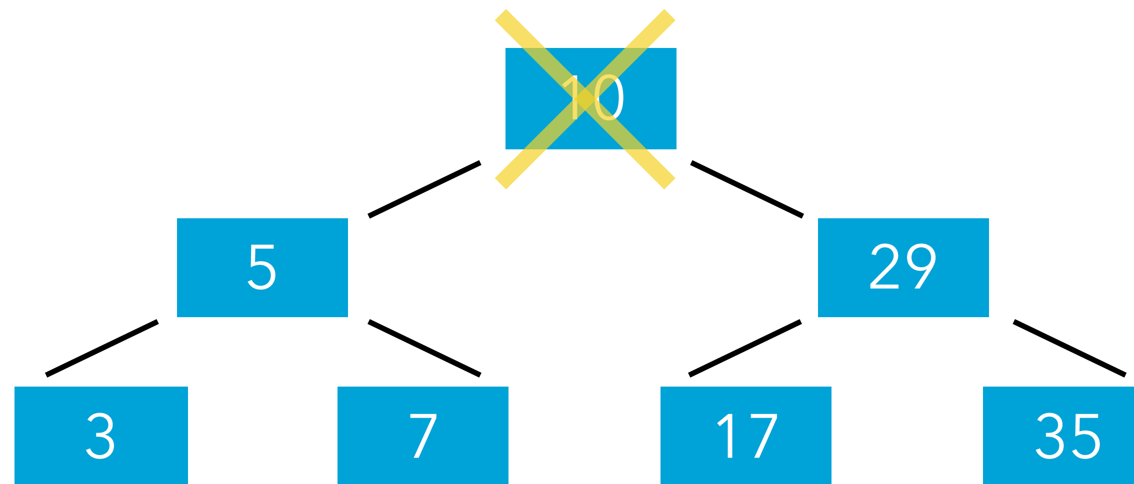
Exemple de suppression

Exemple du cas compliqué : on supprime l'élément 10, qui a deux fils. On le remplace alors par son successeur, 17, dont le nœud peut être supprimé facilement car il ne peut pas avoir de fils gauche.



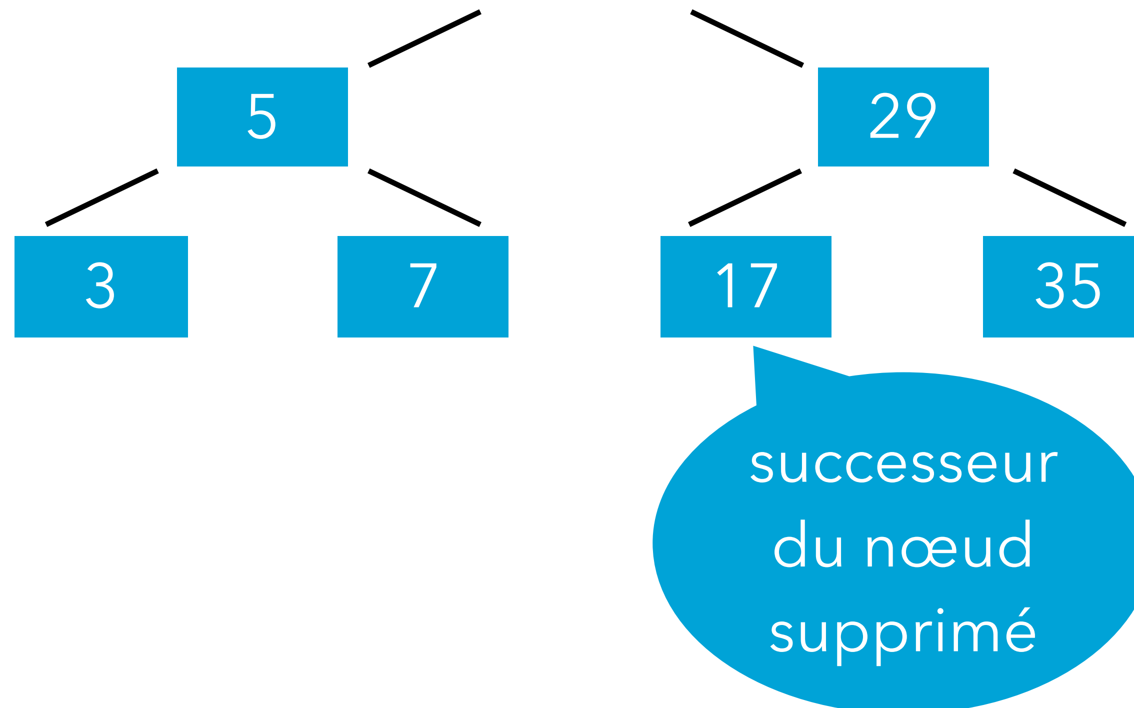
Exemple de suppression

Exemple du cas compliqué : on supprime l'élément 10, qui a deux fils. On le remplace alors par son successeur, 17, dont le nœud peut être supprimé facilement car il ne peut pas avoir de fils gauche.



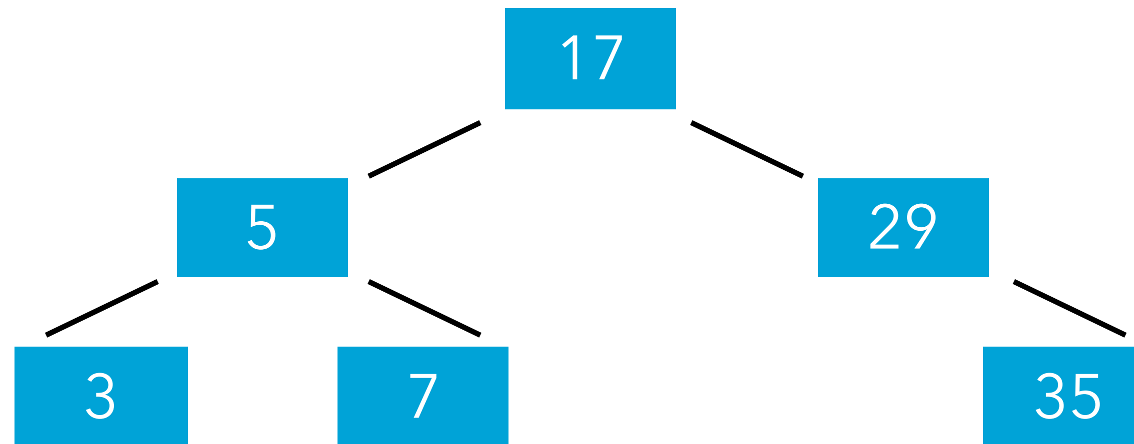
Exemple de suppression

Exemple du cas compliqué : on supprime l'élément 10, qui a deux fils. On le remplace alors par son successeur, 17, dont le nœud peut être supprimé facilement car il ne peut pas avoir de fils gauche.



Exemple de suppression

Exemple du cas compliqué : on supprime l'élément 10, qui a deux fils. On le remplace alors par son successeur, 17, dont le nœud peut être supprimé facilement car il ne peut pas avoir de fils gauche.



Exercice

Pourquoi est-on sûr que le nœud contenant le successeur de l'élément à supprimer n'a pas de fils gauche ?
(Au même titre, d'ailleurs, que l'on est sûr que le nœud contenant son prédécesseur n'a pas de fils droit).

Test d'appartenance

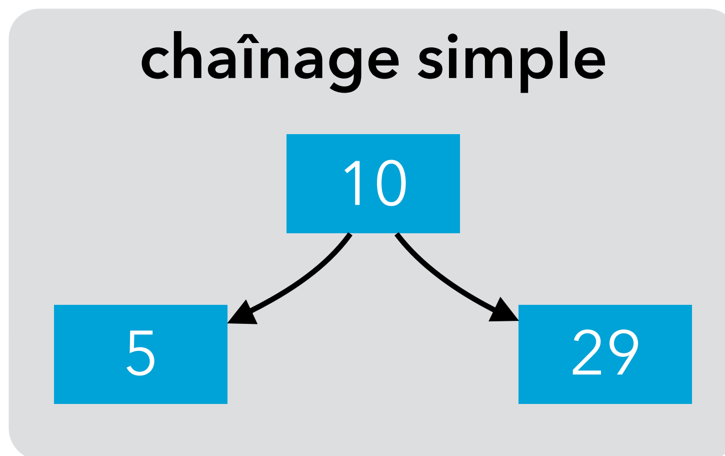
Le test d'appartenance consiste en une simple recherche de l'élément dans l'arbre.
Si celle-ci termine avec succès, l'élément est dans l'ensemble, sinon il n'y est pas.

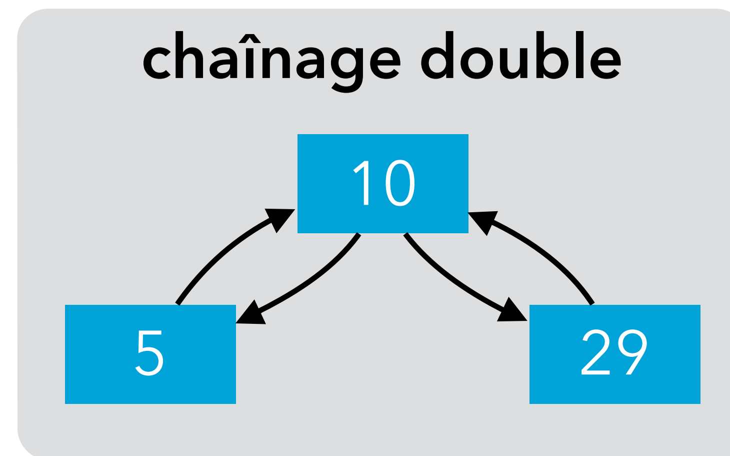
Itération

L'itération sur les éléments d'un arbre binaire de recherche peut tirer parti du fait que les éléments sont ordonnés et les retourner dans l'ordre, du plus petit au plus grand.
Si les nœuds possèdent un lien vers leur parent, l'état de l'itération peut être représenté par une simple référence vers le prochain nœud, comme pour les listes chaînées. Le passage d'un nœud à son successeur est toutefois plus compliqué que pour les listes chaînées…

Chaînage des nœuds

Au même titre qu'une liste peut être simplement ou doublement chaînée, les nœuds d'un arbre peuvent être chaînés dans un seuls sens – d'un parent vers ses fils – ou dans les deux – d'un parent vers ses fils, et des fils vers le parent.
Tout comme pour les listes, le chaînage double permet de simplifier la programmation de certaines opérations.

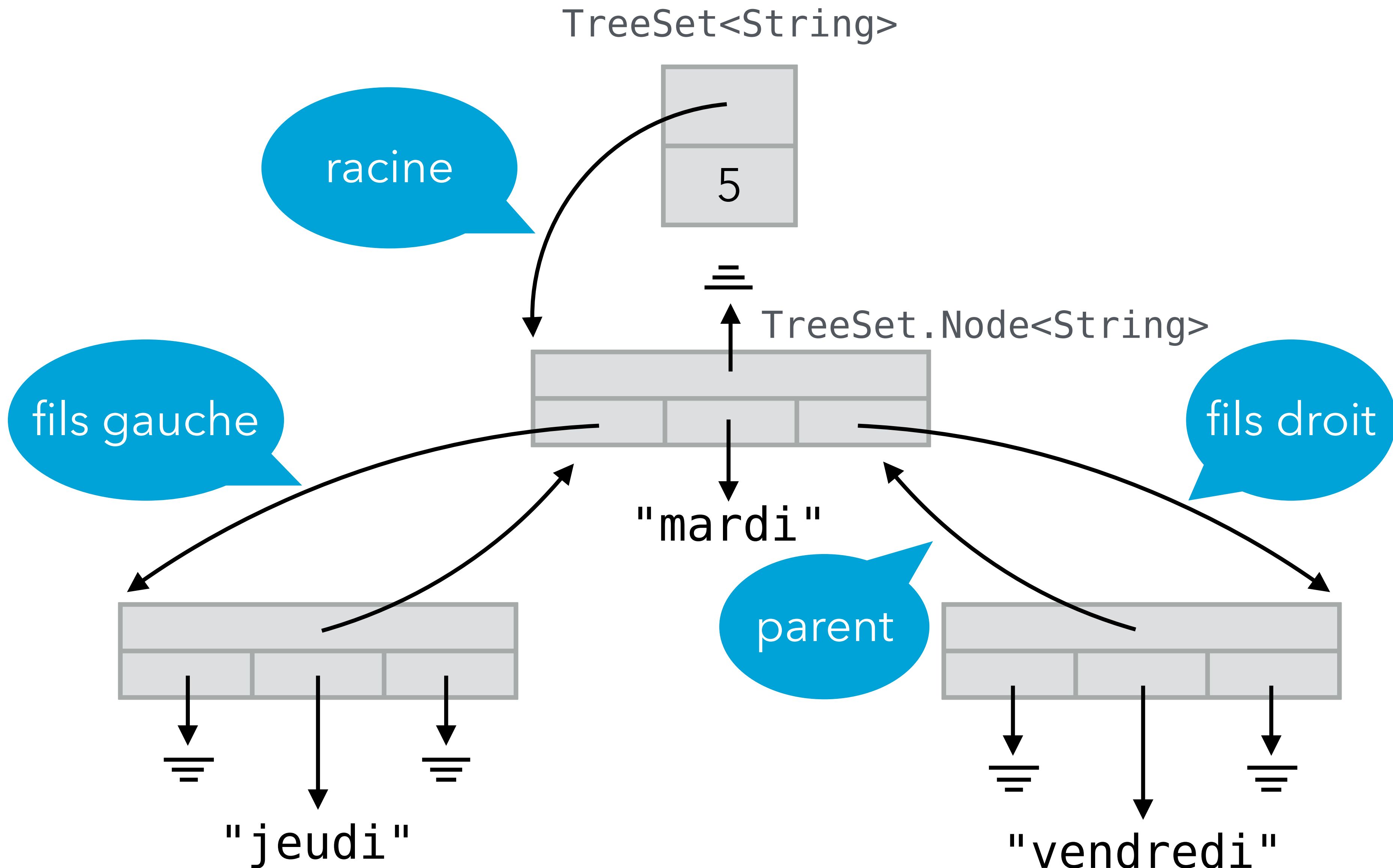




Arbres binaires de recherche en Java

Ensemble par a.b.r.

(Nœuds « doublement » chaînés)



Classe TreeSet (v1)

En première approximation, la classe `TreeSet` ressemble à `LinkedList` mais les nœuds ont un champ de plus, la tête s'appelle la racine, et il n'y a pas de nœud d'en-tête.

```
public final class TreeSet<E>
    implements Set<E> {
  private int size = 0;
  private Node<E> root = null;
  // … méthodes (isEmpty, size, add, etc.)
  private final static class Node<E> {
    public E elem;
    public Node<E> parent;
    public Node<E> smaller, greater;
    // … constructeurs
  }
}
```

Eléments comparables

Un problème se pose très vite lorsqu'on essaie d'écrire les méthodes de la classe `TreeSet` précédente : comment comparer les éléments entre-eux, étant donné qu'on ne sait rien sur leur type (le paramètre de type `E`) ?

Contrairement au hachage, qui est universel en Java – dans le sens où l'existence de la méthode `hashCode` dans la classe `Object` garantit que tout objet est « hachable » – la comparaison n'est disponible que pour les classes qui implémentent l'interface `Comparable`.

Il faudrait donc pouvoir garantir que la classe des éléments implémente bien cette interface…

Comparable (rappel)

Pour mémoire, l'interface `java.lang.Comparable` est une interface comportant une seule méthode, `compareTo`, qui permet de comparer l'objet auquel on l'applique, `this`, à un autre objet du même type :

```
public interface Comparable<T> {
  int compareTo(T that);
}
```

Une classe dont les instances doivent être comparables entre elles implémente cette interface en lui passant son propre type en paramètre. Exemple :

```
class Date implements Comparable<Date> {
  int compareTo(Date that) { … }
}
```

Borne de type

Pour garantir que les éléments d'un ensemble de type `TreeSet` soient comparables entre-eux, il est possible de mettre une **borne** (**supérieure**) sur le paramètre de type `E` de la classe :

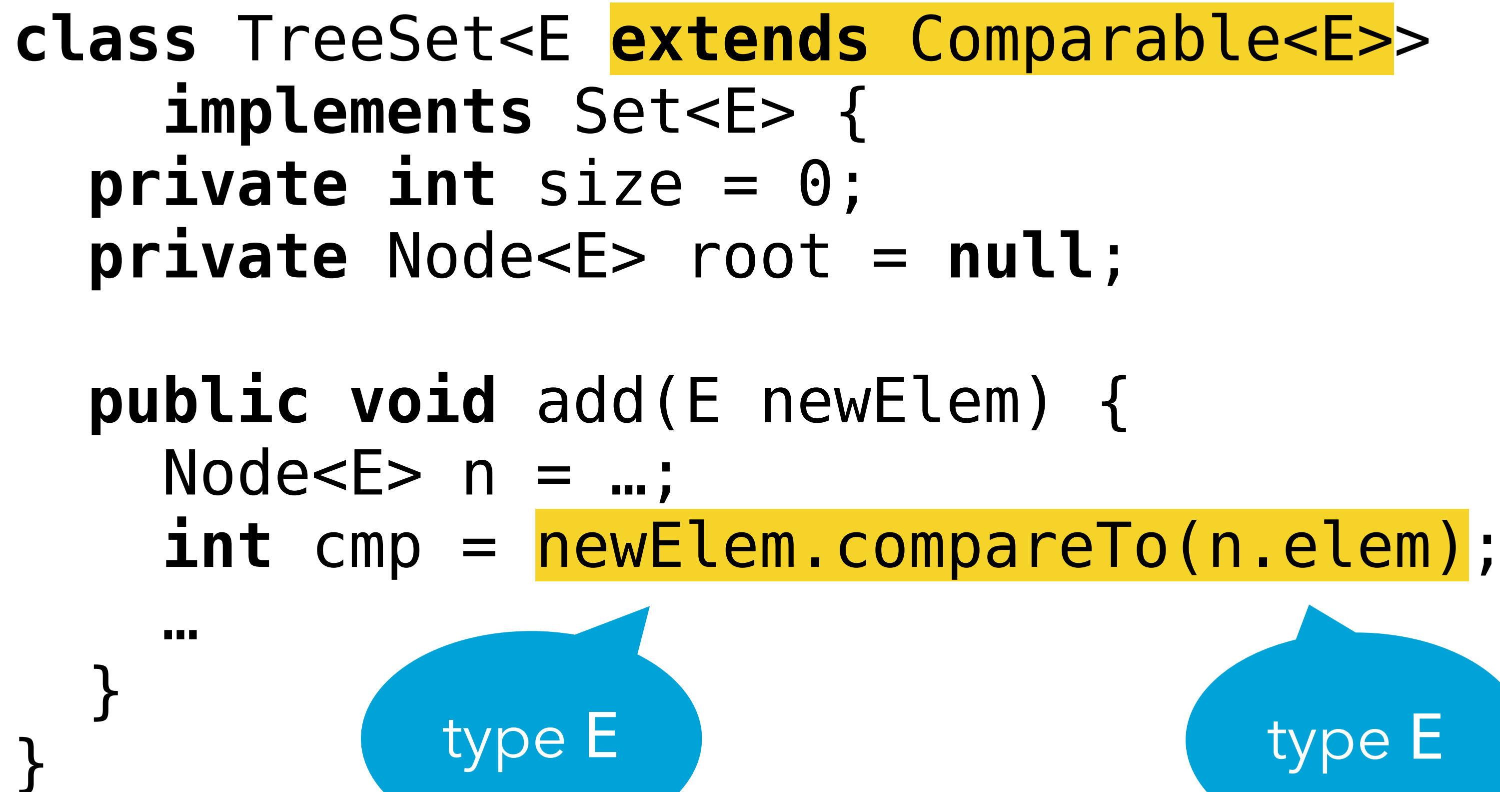
```
class TreeSet<E extends Comparable<E>>
    implements Set<E> { … }
```

Cela fait, seules les sous-types de `Comparable` sont acceptés pour instancier le paramètre `E` :

```
TreeSet<Integer> s1 = …;        // ok
TreeSet<String> s2 = …;         // ok
TreeSet<Object> s3 = …;         // erreur !
TreeSet<List<Integer>> s4 = …; // erreur !
```

Classe TreeSet (v2)

Grâce à la borne, il est possible d'appeler la méthode `compareTo` de `Comparable` pour comparer des objets de type E entre-eux, p.ex. dans la méthode `add` :




```
class TreeSet<E extends Comparable<E>>
      implements Set<E> {
   private int size = 0;
   private Node<E> root = null;

   public void add(E newElem) {
      Node<E> n = …;
      int cmp = newElem.compareTo(n.elem);
      …
   }
}
```

Méthode add

La méthode `add` commence par vérifier si l'arbre est vide. Si tel est le cas, un nouveau nœud contenant l'élément à ajouter est créé et devient la racine de l'arbre.
Si l'arbre n'est pas vide, `add` y recherche l'élément à ajouter.
S'il est présent, il n'y a rien à faire.
S'il est absent, un nouveau nœud est créé pour lui et ajouté comme fils du dernier nœud rencontré lors de la recherche, du côté qui respecte l'invariant – c-à-d à gauche si le nouvel élément est plus petit que celui du dernier nœud, à droite sinon.

Méthode `nodeOrParentFor`

La plupart des méthodes doivent, comme `add`, obtenir le nœud contenant un élément, ou alors son parent si un tel nœud n'existe pas.

Il est donc intéressant d'offrir la méthode auxiliaire `nodeOrParentFor` qui, étant donné un élément, retourne :

- le nœud contenant cet élément s'il est présent dans l'arbre,
- le nœud qui devrait être le parent du nœud contenant cet élément – c-à-d le dernier nœud rencontré lors de la recherche – si l'élément n'est pas présent dans l'arbre.

Cette méthode retourne donc `null` dans un et un seul cas : lorsque l'arbre est vide.

Méthode nodeOrParentFor

```
class TreeSet<E extends Comparable<E>>
    implements Set<E> {
  private int size = 0;
  private Node<E> root = null;

  // … autres méthodes

  private Node<E> nodeOrParentFor(E elem) {
    // ???
  }
}
```

Méthode nodeFor

La méthode auxiliaire `nodeFor` est une variante de `nodeOrParentFor` qui retourne le nœud contenant l'élément donné, ou `null` s'il n'existe pas.

```
class TreeSet<E extends Comparable<E>>
    implements Set<E> {
  private int size = 0;
  private Node<E> root = null;
  // … autres méthode
  private Node<E> nodeFor(E elem) {
    Node<E> n = nodeOrParentFor(elem);
    return n != null && n.elem.equals(elem)
        ? n
        : null;
    }
}
```

Méthode `remove`

La méthode `remove` utilise `nodeFor` pour trouver le nœud contenant l'élément à supprimer. S'il n'existe pas, il n'y a rien à faire.
Dans le cas contraire, si le nœud à supprimer a deux fils, son élément est remplacé par celui de son successeur et le successeur devient le nœud à supprimer.
A ce stade, le nœud à supprimer a au plus un fils, et il suffit de faire en sorte que cet éventuel fils devienne le nouveau fils du nœud à supprimer – comme lors de la suppression d'un nœud d'une liste chaînée.

Méthode successor

La méthode auxiliaire `successor` permet d'obtenir le successeur d'un nœud, c'est-à-dire celui contenant le plus petit élément de l'ensemble qui soit plus grand que son élément à lui.
Cela se fait facilement dans le cas ou le nœud possède un fils droit, puisqu'il suffit alors de parcourir son arête gauche jusqu'à sa fin. Le dernier nœud rencontré est le successeur.

```
private Node<E> successor(Node<E> n) {
  // ???
}
```

Méthode contains

La méthode `contains` appelle simplement `nodeFor` et vérifie que le nœud retourné n'est pas nul :

```
class TreeSet<E extends Comparable<E>>
    implements Set<E> {
  private int size = 0;
  private Node<E> root = null;
  // … autres méthodes
  public boolean contains(E elem) {
    return nodeFor(elem) != null;
  }
}
```

Equilibre des arbres de recherche

Arbres dégénérés

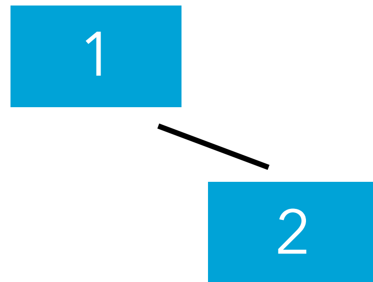
Que se passe-t-il si on insère successivement les entiers de 1 à 4 dans un arbre de recherche initialement vide ?

Arbres dégénérés

Que se passe-t-il si on insère successivement les entiers de 1 à 4 dans un arbre de recherche initialement vide ?

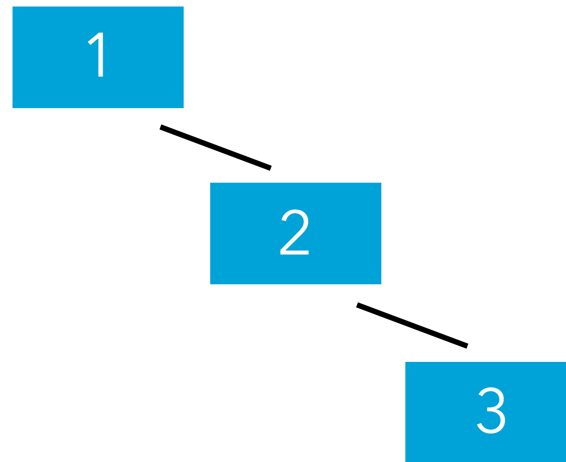
Arbres dégénérés

Que se passe-t-il si on insère successivement les entiers de 1 à 4 dans un arbre de recherche initialement vide ?



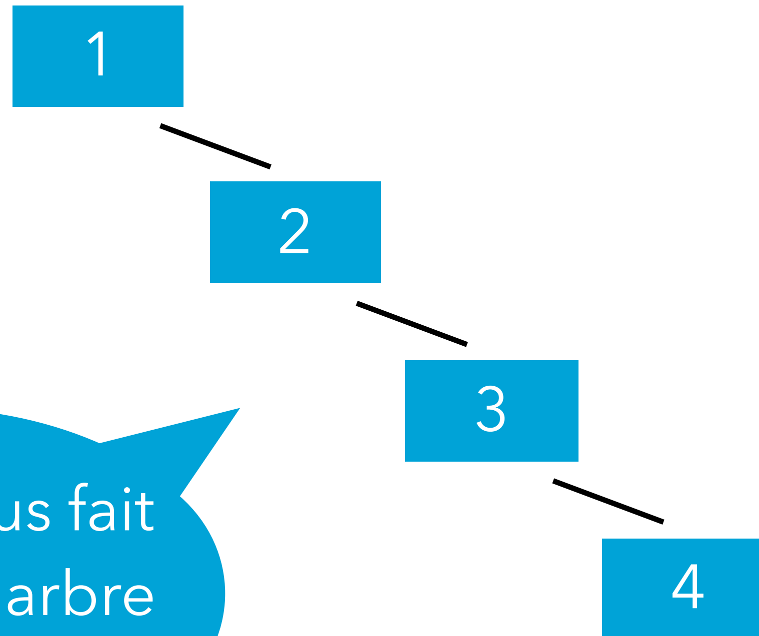
Arbres dégénérés

Que se passe-t-il si on insère successivement les entiers de 1 à 4 dans un arbre de recherche initialement vide ?



Arbres dégénérés

Que se passe-t-il si on insère successivement les entiers de 1 à 4 dans un arbre de recherche initialement vide ?



à quoi vous fait penser cet arbre dégénéré ?

Complexité des opérations

Dans un arbre de recherche, les opérations de base ont une complexité proportionnelle à la hauteur de l'arbre.

Si un arbre binaire de taille n est équilibré, sa hauteur est de l'ordre de $\log_2 n$, donc les opérations de base ont une complexité en $O(\log n)$.

Si un arbre binaire de taille n a dégénéré en liste, il est de hauteur n, donc les opérations de base ont une complexité en $O(n)$.

Conclusion : pour que les opérations de base soient en $O(\log n)$, il faut que les arbres utilisés soient en permanence « assez équilibrés » – une notion que nous ne préciserons pas plus.

Arbres auto-équilibrants

Un arbre de recherche est dit **auto-équilibrant** si ses opérations d'insertion et de suppression conservent l'équilibre de l'arbre.

Il existe plusieurs types d'arbres de recherche auto-équilibrants :

- Les **arbres rouge-noir** (*red-black trees*), inventés en 1972 par Bayer et utilisés dans la bibliothèque Java pour les classes `TreeSet` et `TreeMap`,
- Les **arbres AVL** (*AVL trees*), inventés en 1965 par deux mathématiciens russes, Adelson-Velsky et Landis.

Nous n'examinerons pas ces arbres, mais il est important de connaître leur existence.

Comparateurs

(digression)

Ordre naturel

Notre classe `TreeSet` requiert que la classe des éléments de l'ensemble implémente l'interface `Comparable`. Dans la bibliothèque Java, la plupart des classes dont les éléments sont naturellement ordonnés implémentent cette interface (p.ex. `Integer`, `Double`, `String`, etc.).
L'ordre défini par la méthode `compareTo` de `Comparable` est appelée l'**ordre naturel** de la classe en question.

Ordre externe

Dans certains cas, il peut être intéressant d'utiliser un autre ordre que l'ordre naturel, ou alors de définir un ordre pour une classe qui n'a pas d'ordre naturel. On dit alors qu'on veut utiliser un **ordre externe** pour ordonner des éléments. La bibliothèque Java offre la notion de comparateur pour représenter les ordres externes.

Comparateur

Un **comparateur** est un objet sachant comparer deux objets d'un même type. Cette notion est naturellement exprimée dans la bibliothèque Java au moyen d'une interface, nommée `Comparator` (du paquetage `java.util`) et définie ainsi :

```
public interface Comparator<T> {
  int compare(T o1, T o2);
}
```

L'entier retourné par la méthode `compare` est négatif si le comparateur considère le premier objet inférieur au second, nul s'il les considère égaux, et positif s'il considère le premier supérieur au second.

Comparateur d'entiers

Par exemple, le comparateur ci-dessous compare deux entiers selon l'ordre inverse de l'ordre habituel, c-à-d qu'il considère un entier plus petit qu'un autre s'il est plus grand, et inversement :

```
Comparator<Integer> invIntComparator =
  new Comparator<Integer>() {
    public int compare(Integer i1,
                       Integer i2) {
      return Integer.compare(i2, i1);
    }
  };
```

Notez que ce comparateur est une instance d'une classe anonyme qui implémente l'interface `Comparator`.

Comparateurs et TreeSet

Notre classe `TreeSet` exige que les éléments de l'ensemble aient un ordre naturel.

La classe `TreeSet` de la bibliothèque Java est plus flexible et peut accepter à la construction un comparateur à utiliser pour comparer les éléments. Exemple d'utilisation :

```
import java.util.TreeSet;
import java.util.Set;
Set<Integer> s =
  new TreeSet<>(invIntComparator);
```

L'itérateur de l'ensemble `s` fournit ainsi les entiers en ordre décroissant.

`Comparator`≠`Comparable`

Attention, malgré les similarités, `Comparator` et `Comparable` ont des caractéristiques très différentes !

- L'interface `Comparable` est implémentée directement par la classe des objets à comparer. Sa méthode `compareTo` prend un seul argument (explicite), le second étant implicitement `this`.
- L'interface `Comparator` est implémentée par un objet sans lien direct avec la classe des objets à comparer – souvent une instance d'une classe anonyme. Sa méthode `compare` prend deux arguments.