Collections : Ensembles (I)

Théorie et pratique de la programmation

Michel Schinz – 2013-03-04

Ensemble (rappel)

Un **ensemble** (*set*) est une collection non-ordonnée d'éléments qui n'admet pas les éléments dupliqués.
Les opérations principales sur un ensemble sont :

- l'ajout d'un élément,
- la suppression d'un élément,
- le test d'appartenance d'un élément,
- le parcours des éléments, dans un ordre (souvent) quelconque.

Exemple : ensemble des nombres premiers compris entre 0 et 1000.

Interface

Une interface simple pour les ensembles de valeurs quelconques peut se présenter ainsi :

```
interface Set<E> {
  boolean isEmpty();
  int size();
  void add(E newElem);
  void remove(E elem);
  boolean contains(E elem);
}
```

Exemple d'utilisation

```
Set<Integer> primes = new …;
for (int i = 0; i <= 1000; ++i) {
  if (isPrime(i)) {
    primes.add(i);
  }
}
```

```
System.out.println("Il y a "+ primes.size()
  +" nombres premiers entre 0 et 1000");
System.out.println("N.p. inférieurs à 20 :");
for (int i = 0; i < 20; ++i) {
  if (primes.contains(i)) {
    System.out.println(i);
  }
}
```

Mises en œuvre

Nous examinerons deux mises en œuvre du concept d'ensemble :

- les **arbres (binaires) de recherche**, qui supposent que les éléments de l'ensemble peuvent être ordonnés,
- les **tables de hachage**, qui supposent que les éléments de l'ensemble peuvent être « hachés » en un entier bien distribué.

La complexité des opérations de base diffèrent entre ces deux mises en œuvre, de même que l'utilisation mémoire.

Mise en œuvre 1 : arbre de recherche

Liste comme ensemble

Une première idée (naïve) pour représenter un ensemble d'éléments consiste à utiliser une liste chaînée.

Malheureusement, aucune opération n’est efficace :

- ajout et suppression : O(n)
- test d’appartenance : O(n)

Note : l'ajout est en O(n) même si l'ajout dans une liste chaînée est en O(1), car il faut parcourir tous les éléments de la liste pour s'assurer que celui qu'on ajoute ne s'y trouve pas déjà !

Tableau comme ensemble

Une autre idée – presque aussi naïve – consiste à utiliser un tableau pour représenter un ensemble.
Malheureusement, les opérations de base ne sont pas plus efficaces qu'avec les listes !
Toutefois, s'il est possible d'ordonner les éléments, une technique intéressante peut être utilisée...

Recherche dichotomique

Pour trouver si un élément donné est présent dans un tableau *trié*, on peut utiliser la **recherche dichotomique**. L'idée est de séparer le tableau en deux moitiés égales, puis de regarder dans quelle moitié l'élément recherché devrait être. Ensuite, on recommence la recherche dans cette moitié, jusqu'à tomber sur l'élément recherché, ou un tableau vide si cet élément n'est pas dans l'ensemble.

Recherche dichotomique

Exemple : on recherche si 17 apparaît dans le tableau trié d'entiers ci-dessous.

3	5	7	10	17	29	35

Recherche dichotomique

Exemple : on recherche si 17 apparaît dans le tableau trié d'entiers ci-dessous.

3	5	7	10	17	29	35

Recherche dichotomique

Exemple : on recherche si 17 apparaît dans le tableau trié d'entiers ci-dessous.

3	5	7	10	17	29	35

17	29	35

Recherche dichotomique

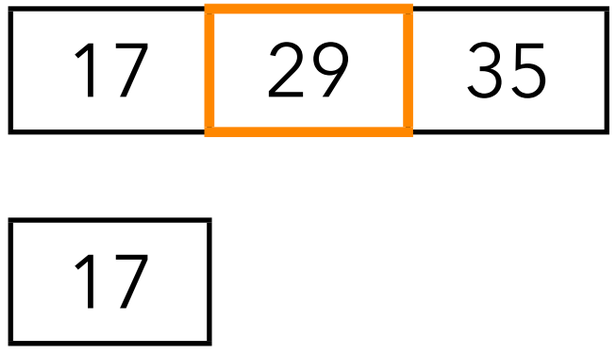
Exemple : on recherche si 17 apparaît dans le tableau trié d'entiers ci-dessous.

3	5	7	10	17	29	35

17	29	35

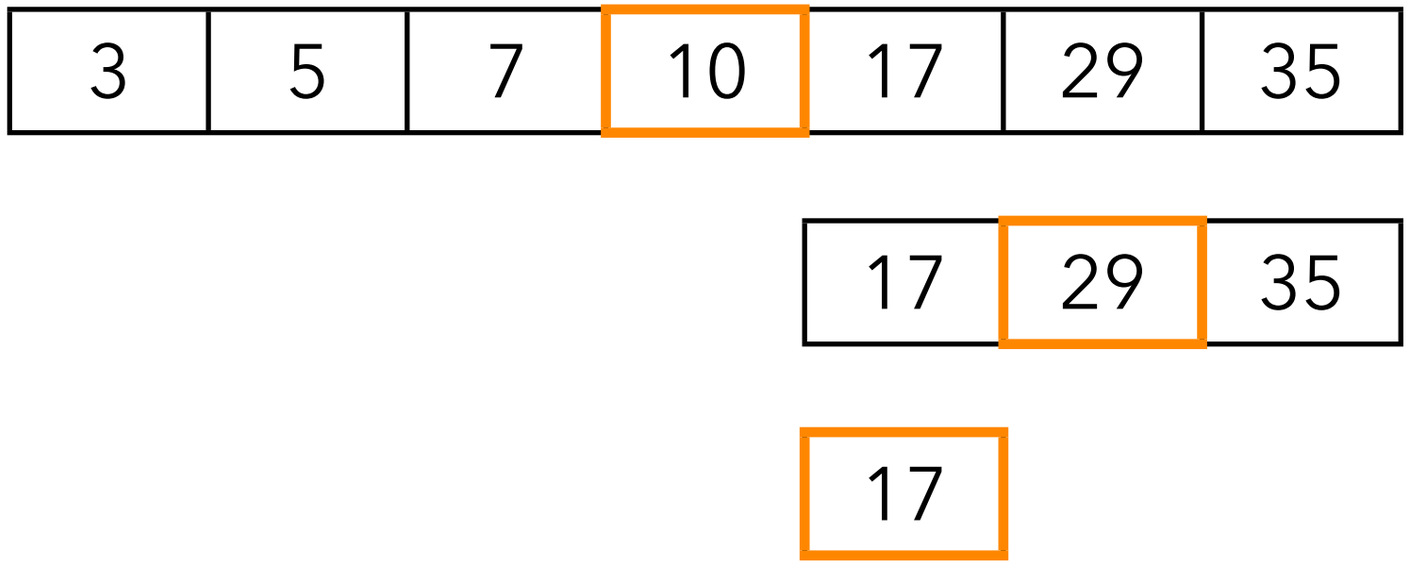
Recherche dichotomique

Exemple : on recherche si 17 apparaît dans le tableau trié d'entiers ci-dessous.



Recherche dichotomique

Exemple : on recherche si 17 apparaît dans le tableau trié d'entiers ci-dessous.



Recherche dichotomique

```
int search(int[] a, int s, int b, int e) {
  if (b > e) {
    return -1;
  } else {
    int mid = (b + e) / 2;
    int midE = a[mid];
    if (s < midE)
      return search(a, s, b, mid - 1);
    else if (midE < s)
      return search(a, s, mid + 1, e);
    else return mid;
  }
}
```

intervalle vide : pas trouvé

recherche dans la moitié sup.

trouvé !

recherche dans la moitié inf.

Recherche dichotomique

La technique de la recherche dichotomique permet de mettre en œuvre les ensembles au moyen de tableaux triés.
Le test d'appartenance devient ainsi moins cher :

- ajout et suppression : O(n) [toujours encore]
- test d'appartenance : O(log n)

Attention : on doit pouvoir ordonner les éléments !

Arbre de recherche

La recherche dichotomique nous permet de rendre le test d’appartenance plus efficace. Malheureusement, en stockant les éléments dans un tableau, l’insertion et la suppression sont encore chères.

Peut-on définir une structure de données qui permette un test d’appartenance aussi efficace que les tableaux triés, mais qui soit meilleure pour l’insertion et la suppression ?
Oui, l'arbre de recherche !

Arbre de recherche

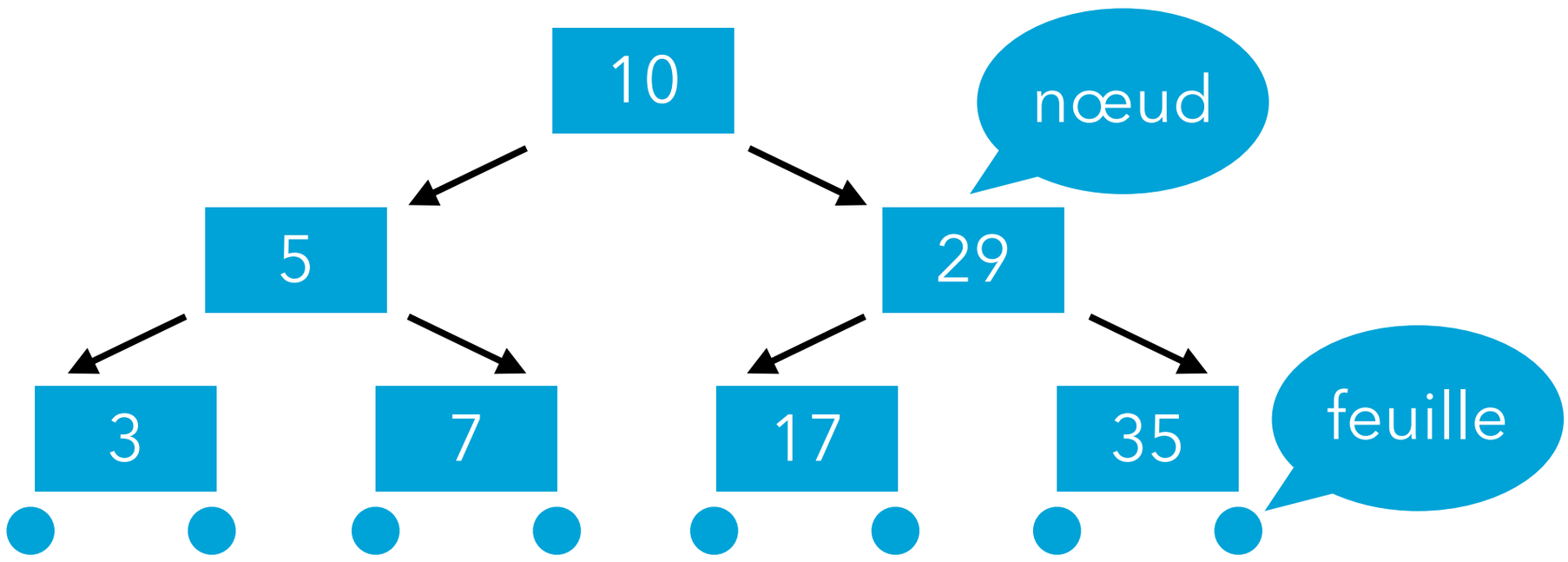
Un **arbre (binaire) de recherche** (*binary search tree*) est un arbre composé de **nœuds** ayant chacun deux fils, et de **feuilles**.

A chaque nœud est associé un élément et l'arbre est organisé de manière à ce que l'invariant (c-à-d la condition) 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.

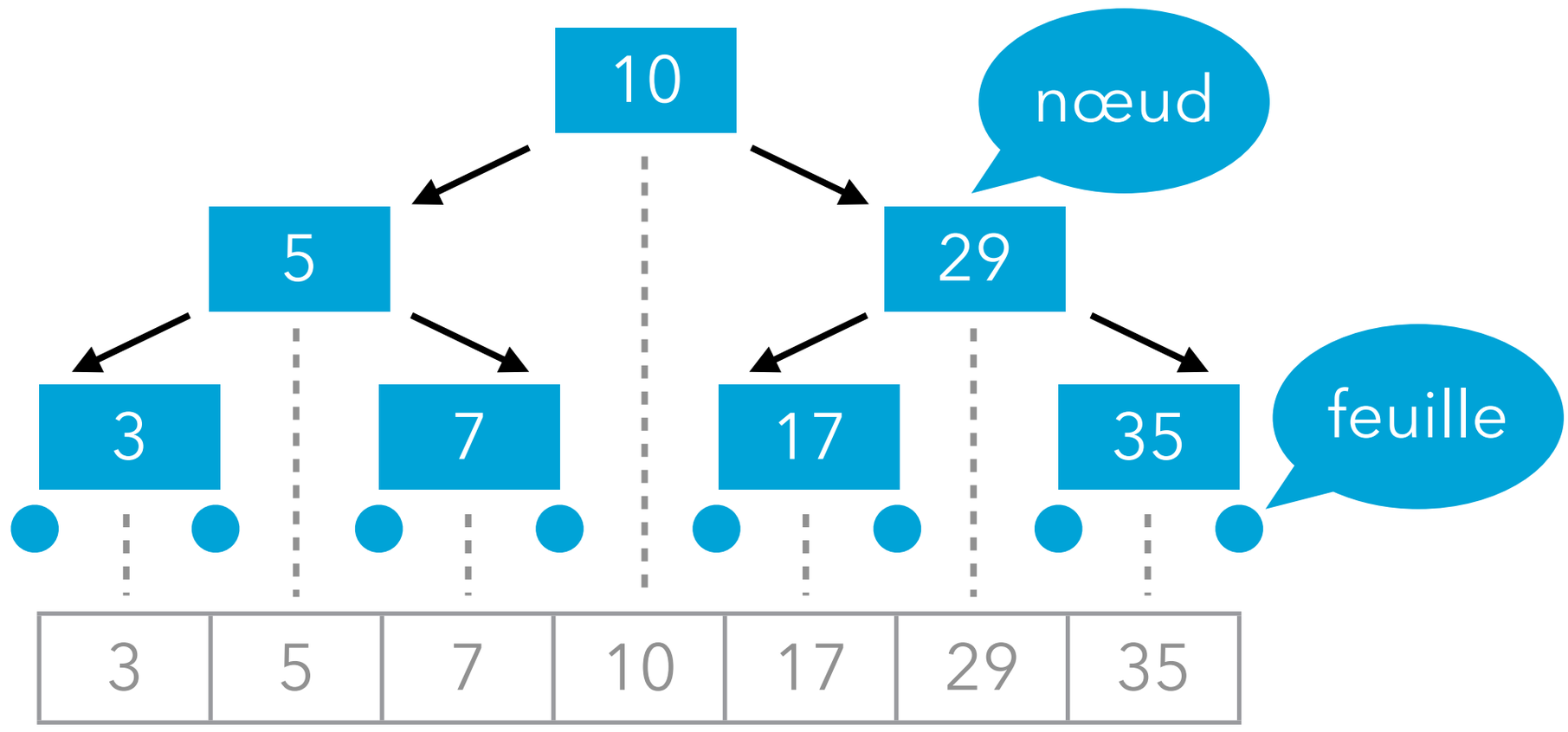
Arbre de recherche

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

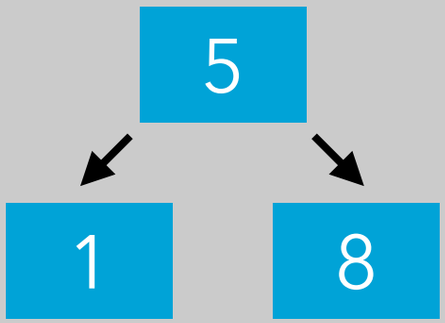
Invariant (rappel) : pour tout nœud, tous les éléments du sous-arbre gauche sont strictement plus petits que celui du nœud, ceux du sous-arbre droite strictement plus grands.



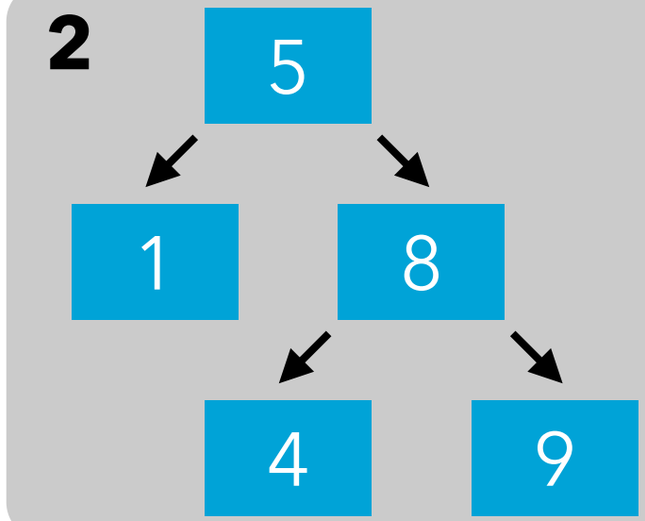
Exercice

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.

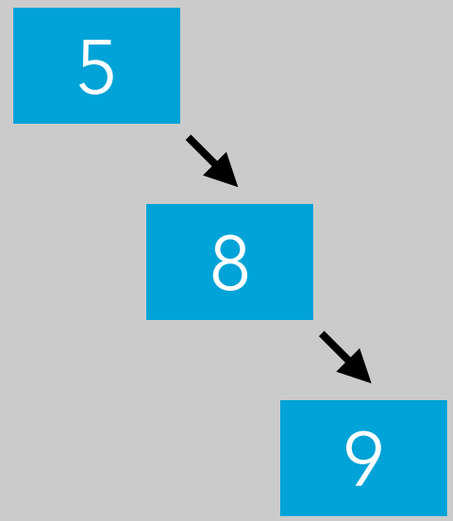
1



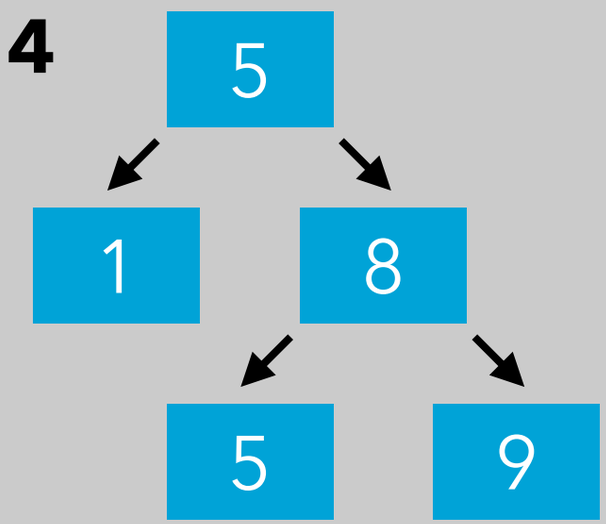
2



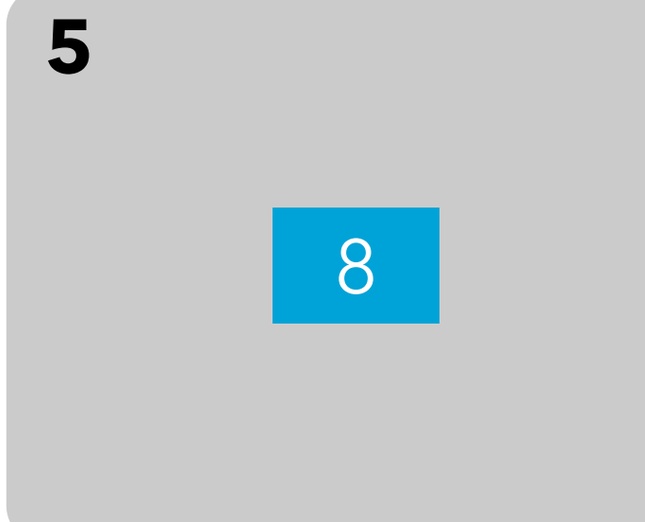
3



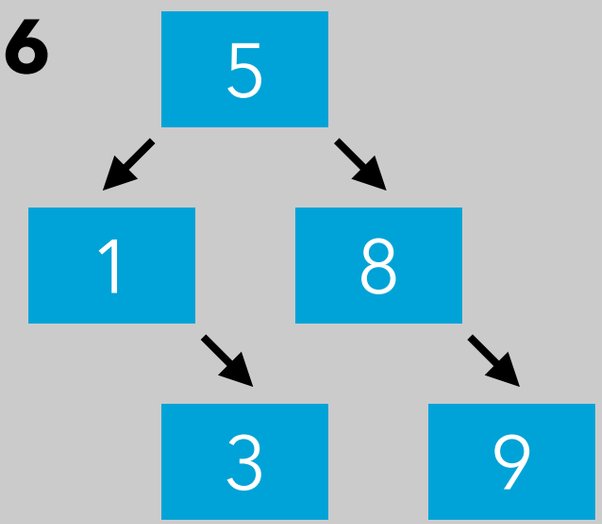
4



5

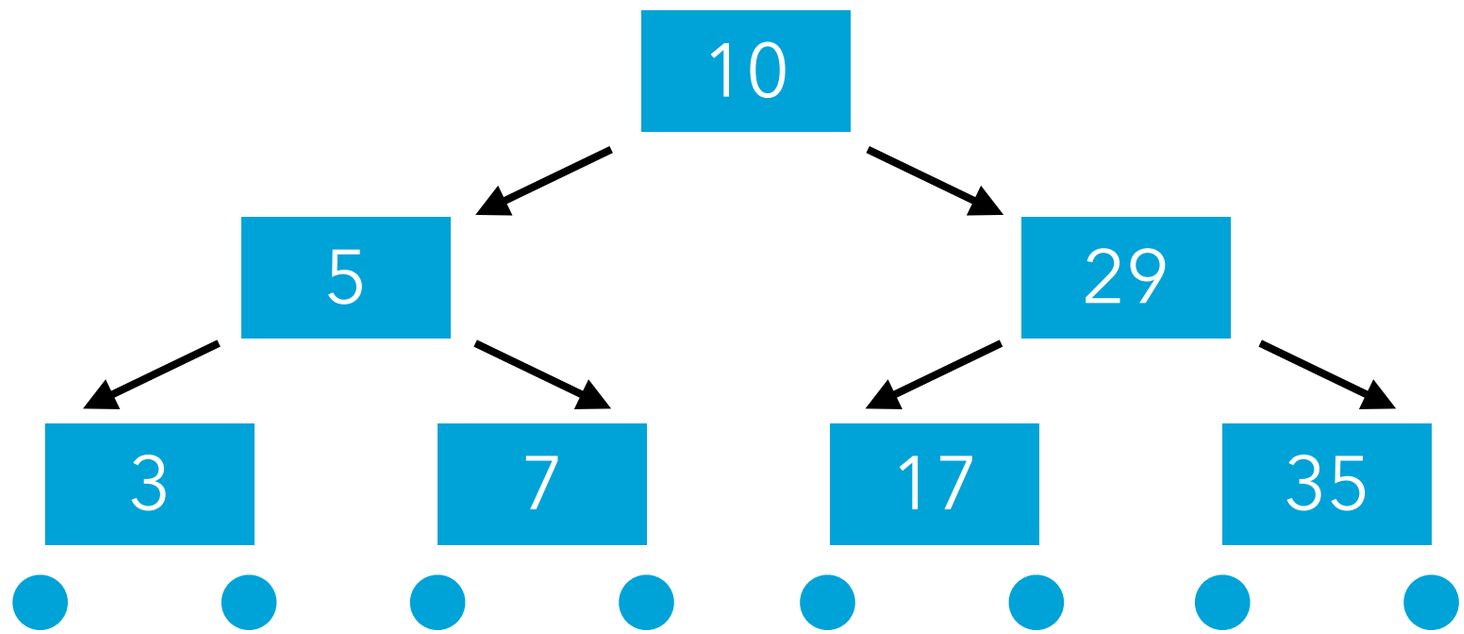


6



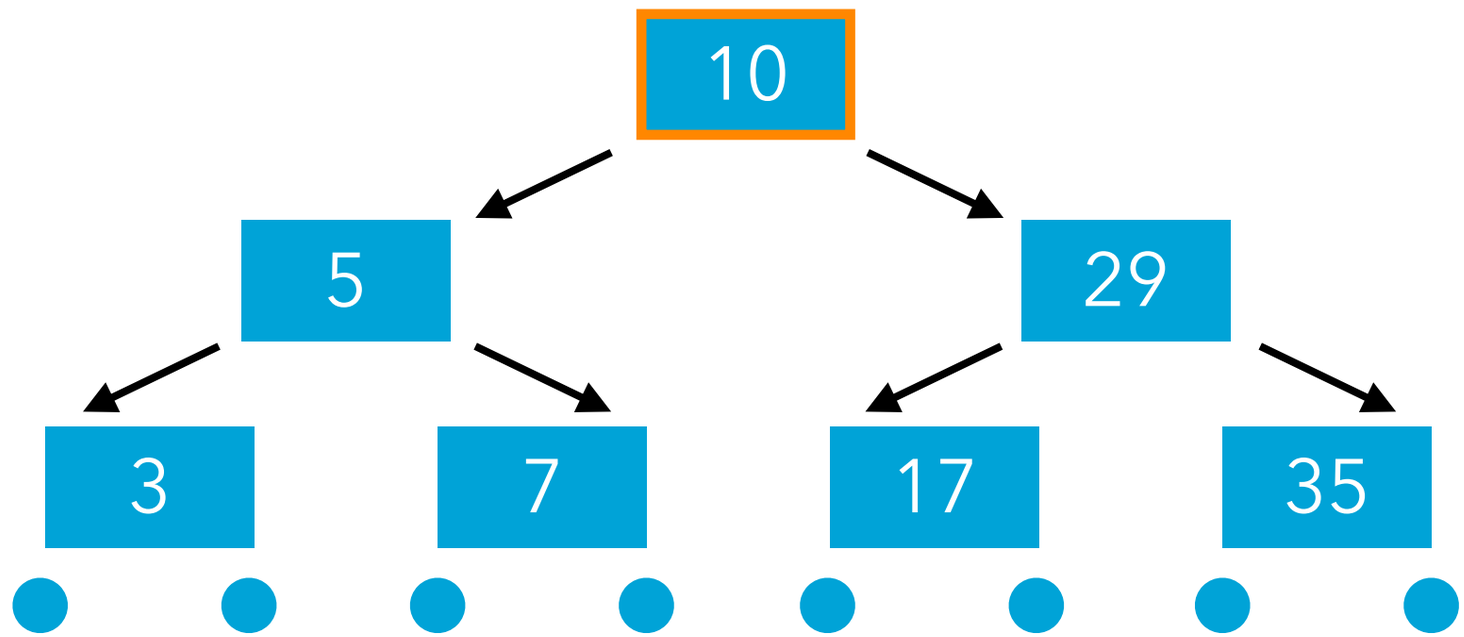
Recherche d'un élément

Exemple : on recherche l'élément 17.



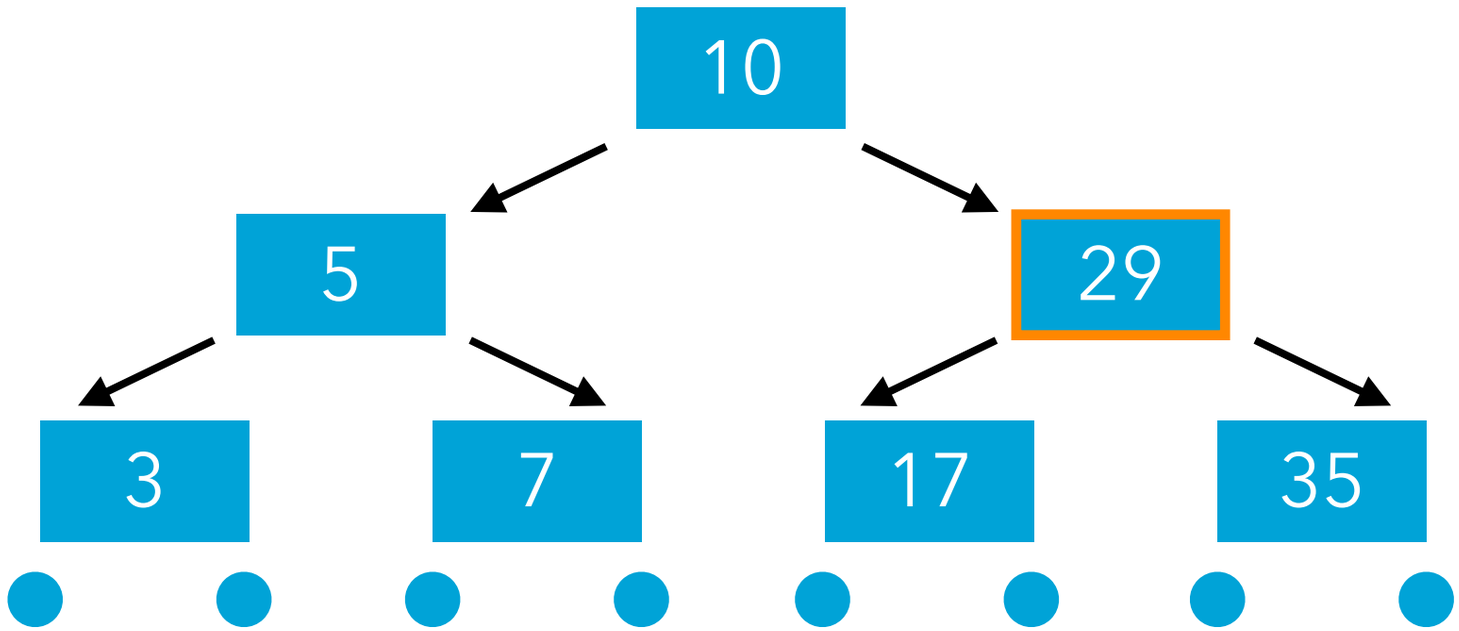
Recherche d'un élément

Exemple : on recherche l'élément 17.



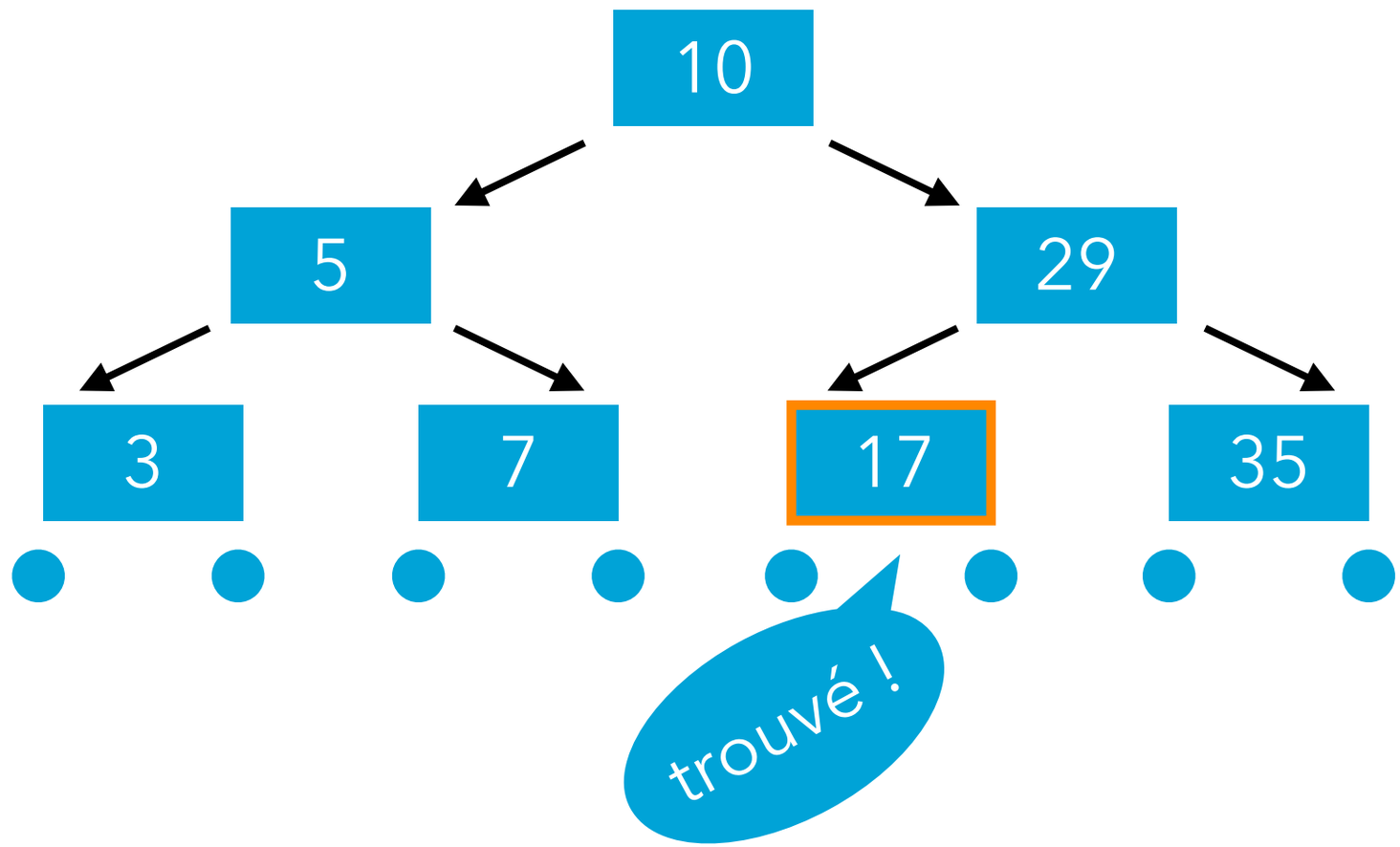
Recherche d'un élément

Exemple : on recherche l'élément 17.



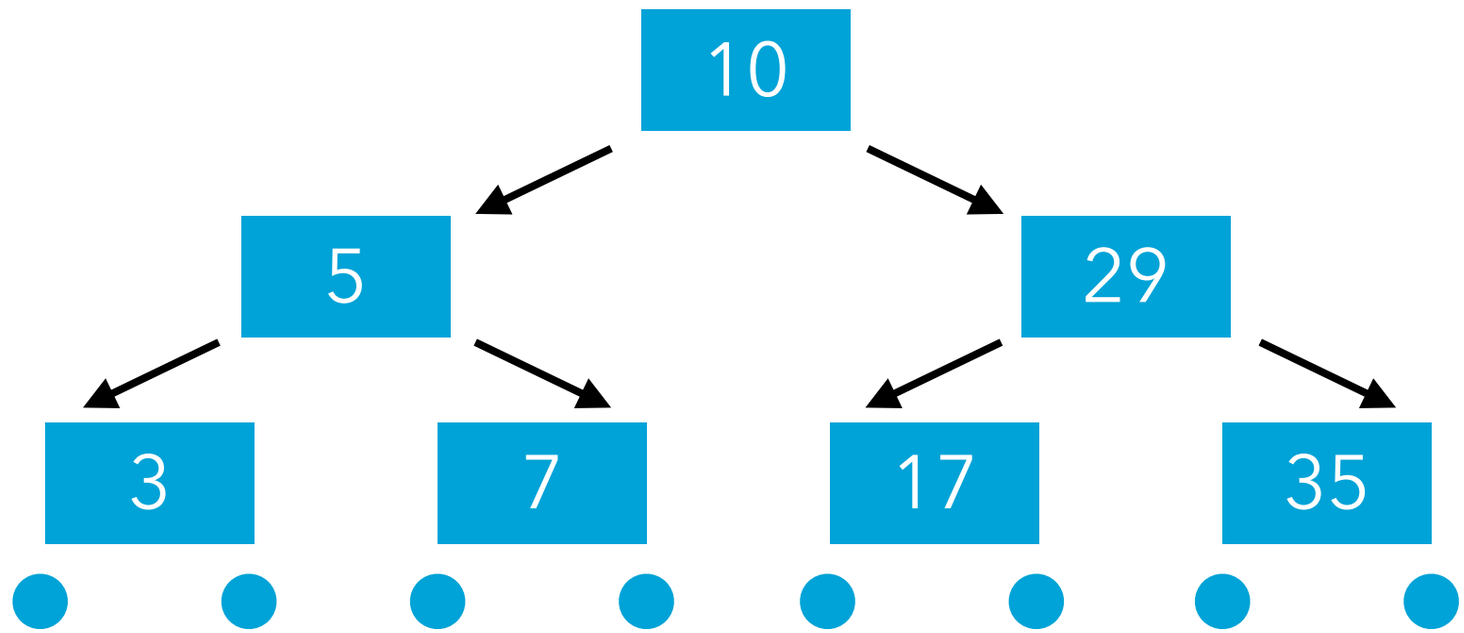
Recherche d'un élément

Exemple : on recherche l'élément 17.



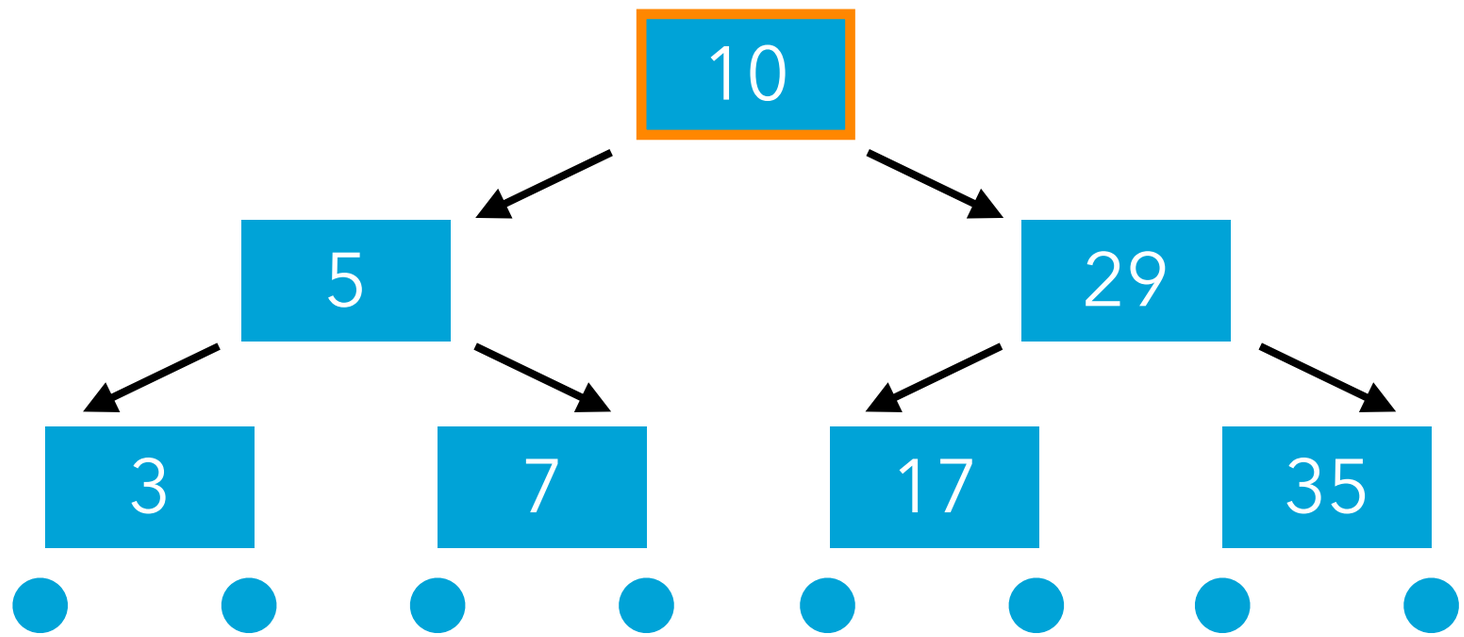
Recherche d'un élément

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



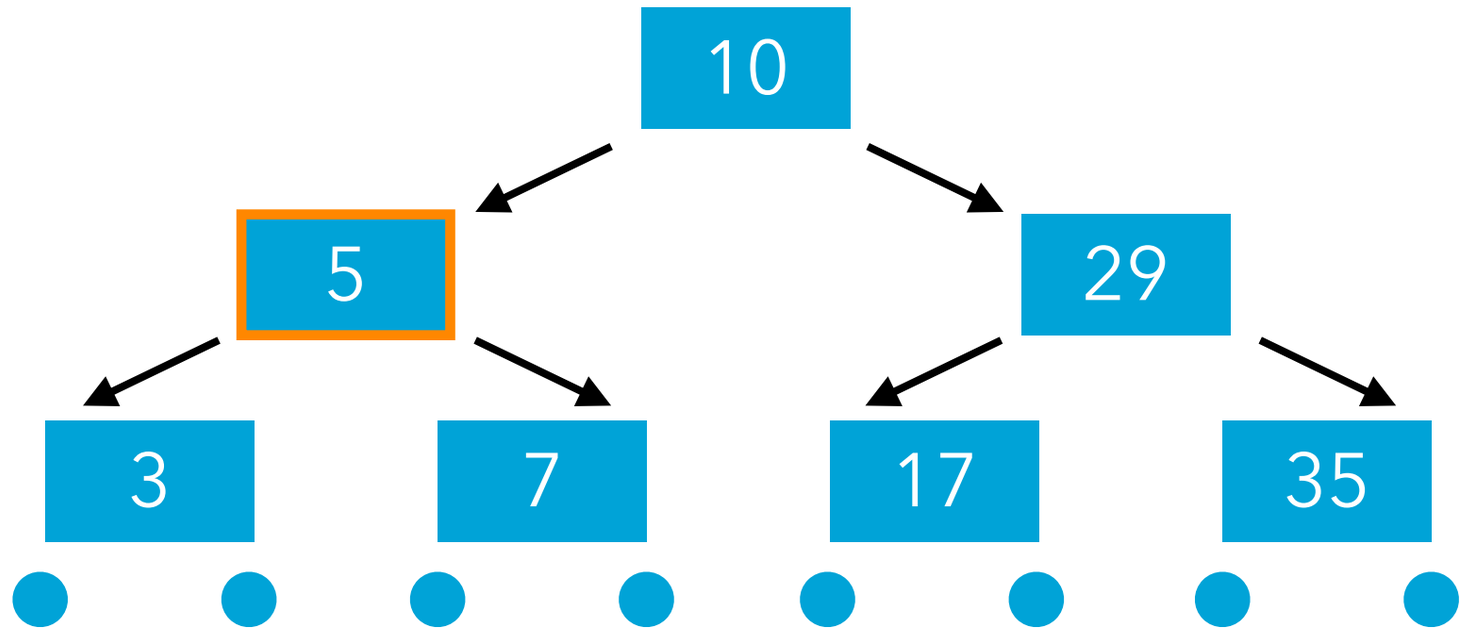
Recherche d'un élément

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



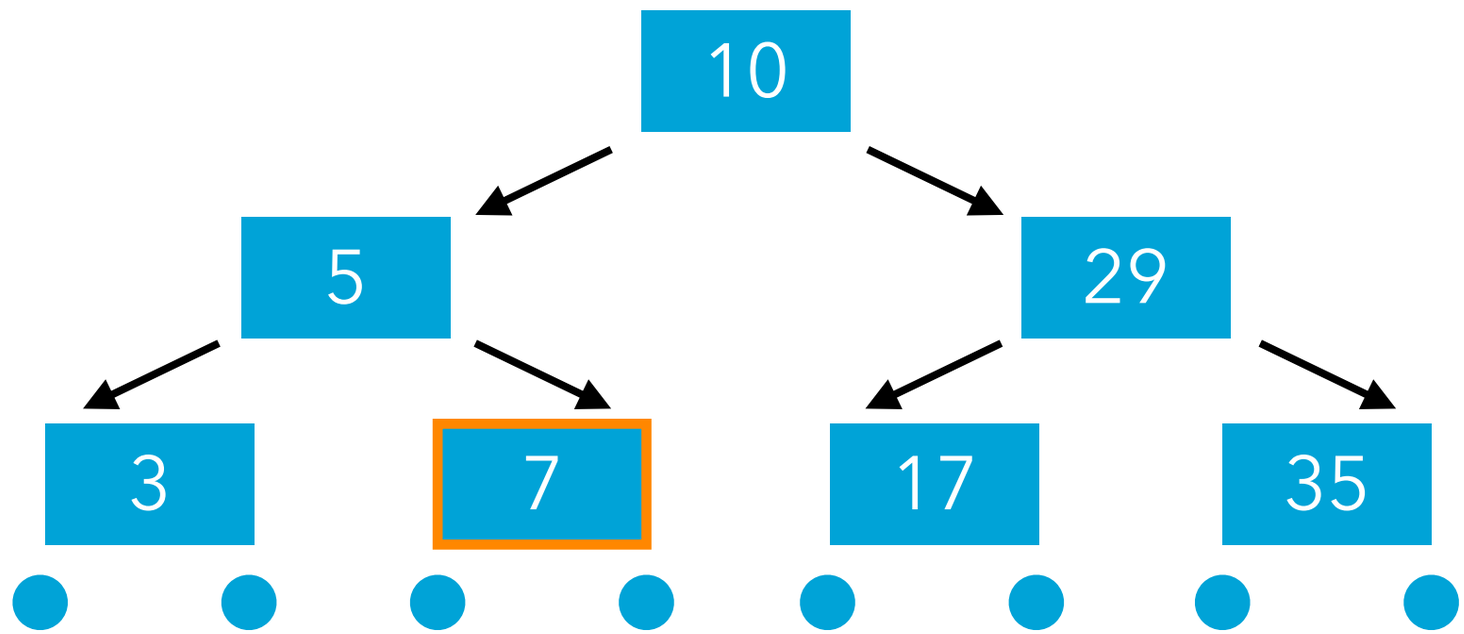
Recherche d'un élément

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



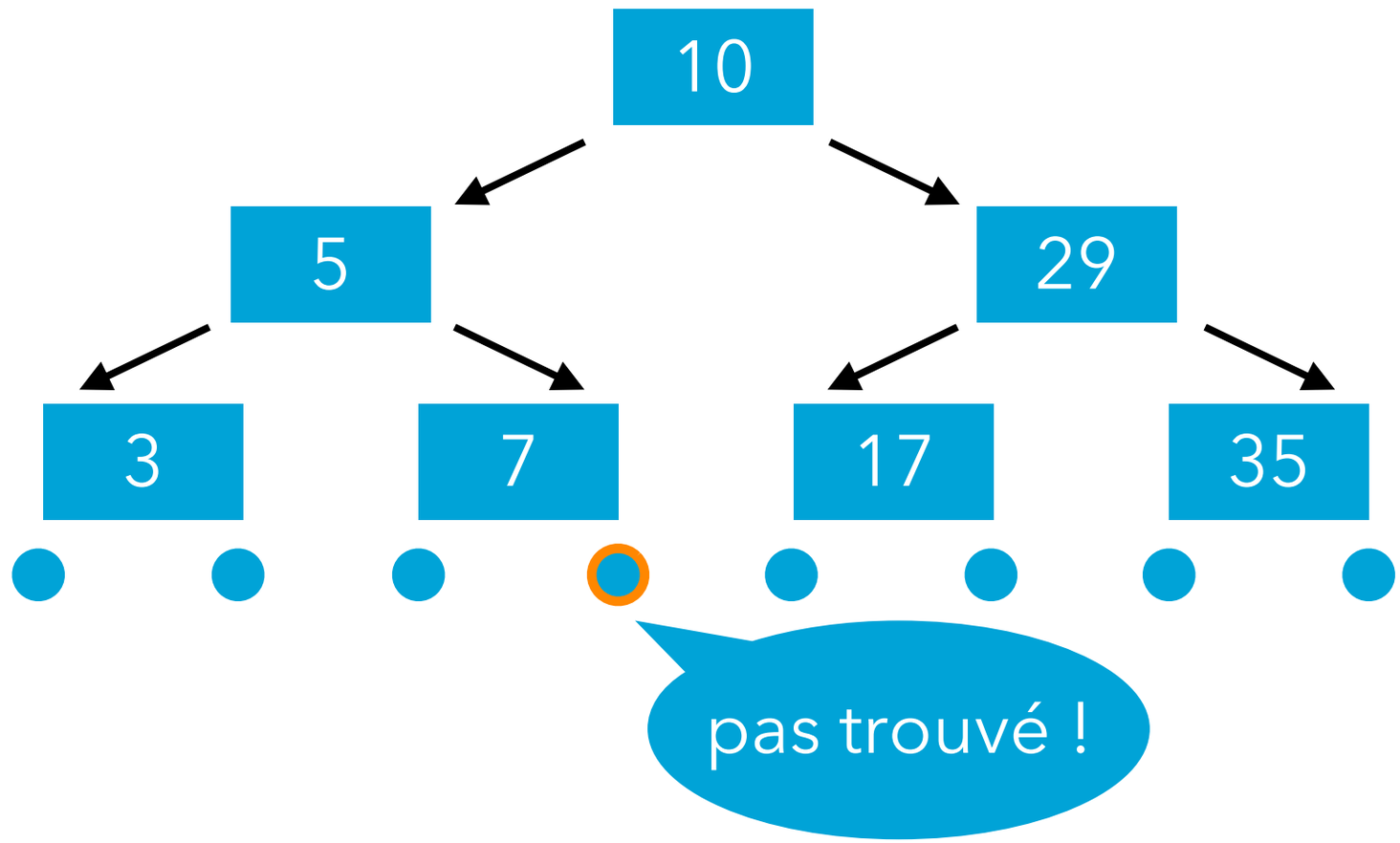
Recherche d'un élément

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



Recherche d'un élément

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



Arbres de recherche en Java

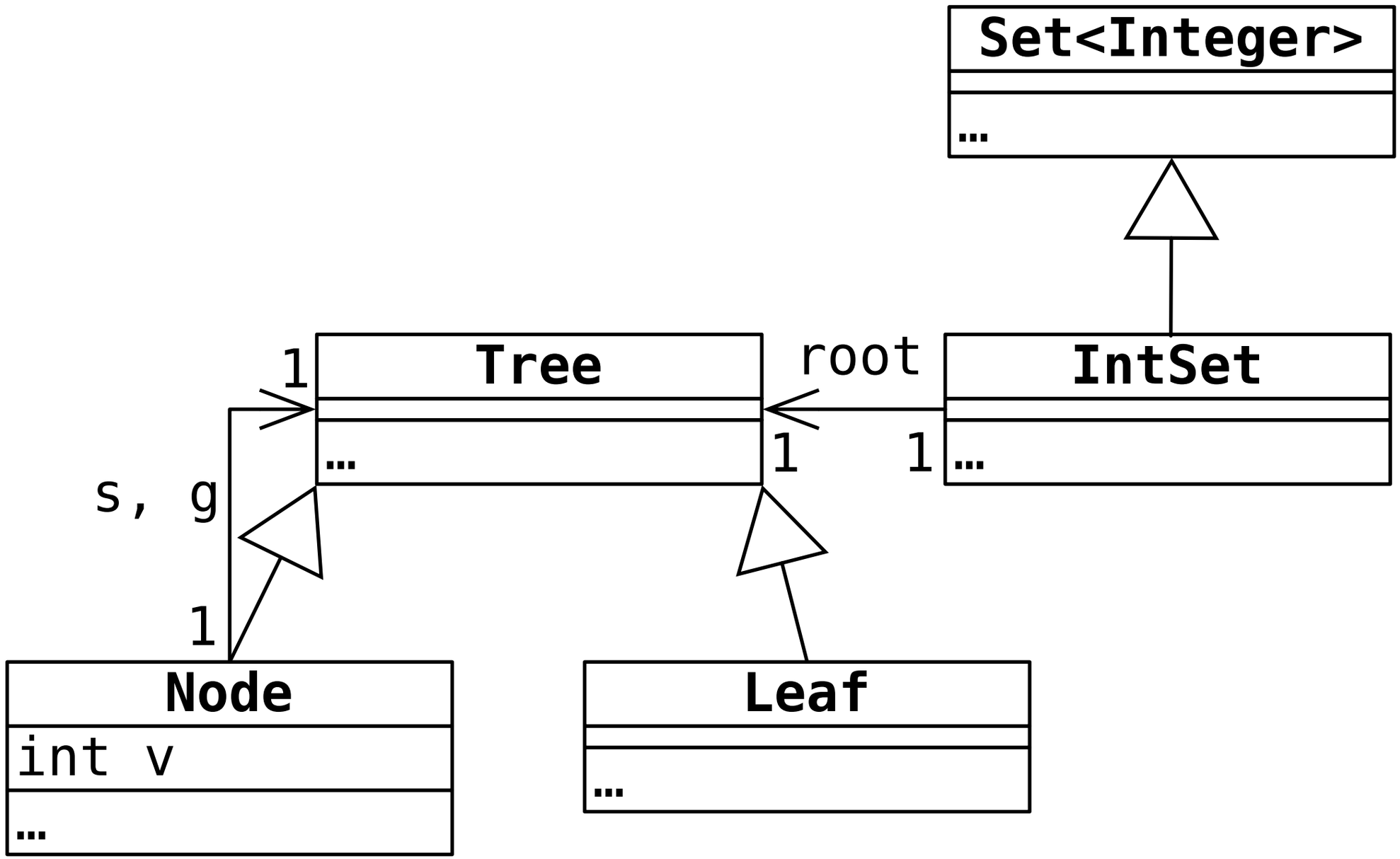
Représentation des arbres

Pour représenter les arbres de recherche en Java, on peut utiliser l'ensemble de classes suivant :

- une interface `Tree` pour les arbres,
- une sous-classe `Node` pour les nœuds,
- une sous-classe `Leaf` pour les feuilles.

Au moyen de ces classes, il devient ensuite aisé de définir, par exemple, une classe `IntSet` représentant les ensembles d'entiers.

Diagramme de classe



Code (1)

```
interface Tree {
   …
   public boolean contains(int elem);
}
```

```
class Leaf implements Tree {
   …
   public boolean contains(int elem) {
      return false;
   }
}
```

Code (2)

```
class Node implements Tree {
  private final int v;
  private Tree s, g;

  public Node(Tree s, int v, Tree g) {
    this.s = s;
    this.v = v;
    this.g = g;
  }

  …
  public boolean contains(int elem) { ??? }
}
```

Code (3)

```
class IntTreeSet implements Set<Integer> {
  private Tree root = new Leaf();
  …
  public boolean contains(Integer elem) {
    return root.contains(elem);
  }
}
```

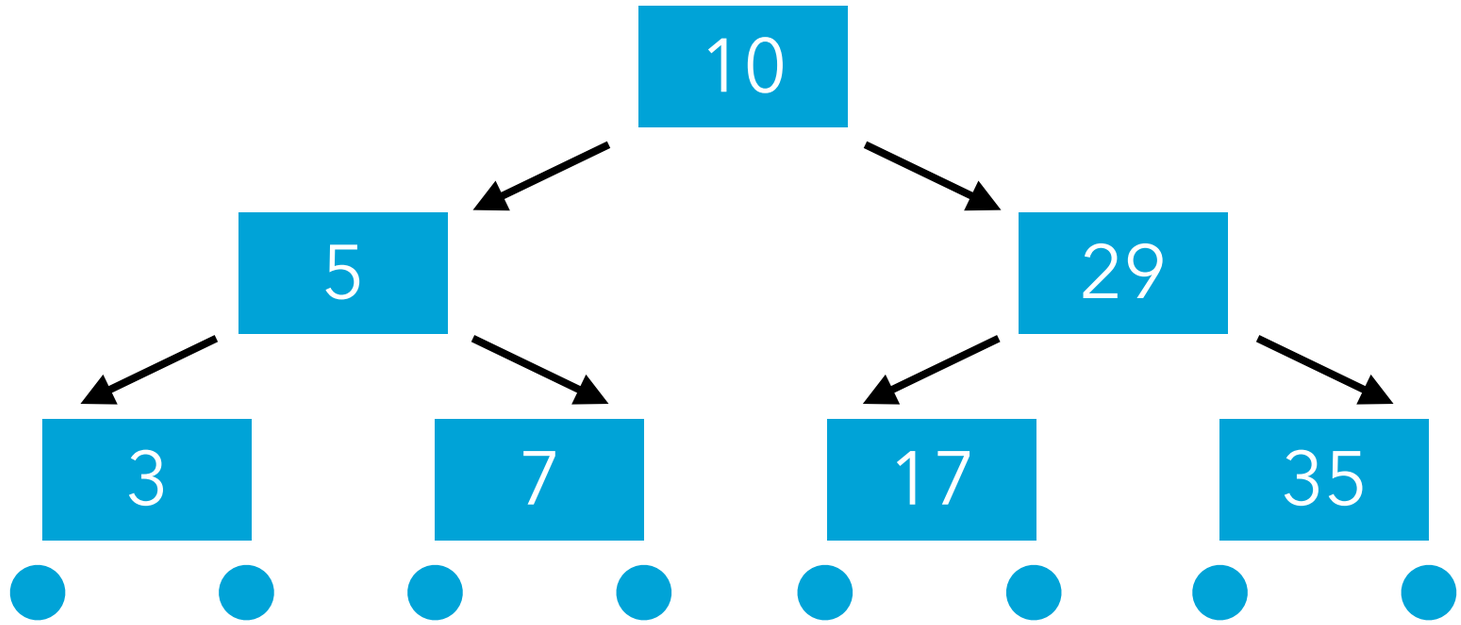
Insertion dans un a.b.r.

Algorithme d'insertion

L’insertion d’un élément dans un arbre de recherche est très similaire à la recherche.
La seule différence concerne la manière dont on traite les feuilles : lorsqu'on en rencontre une durant une insertion, on la remplace par un nouveau nœud contenant l’élément à insérer.

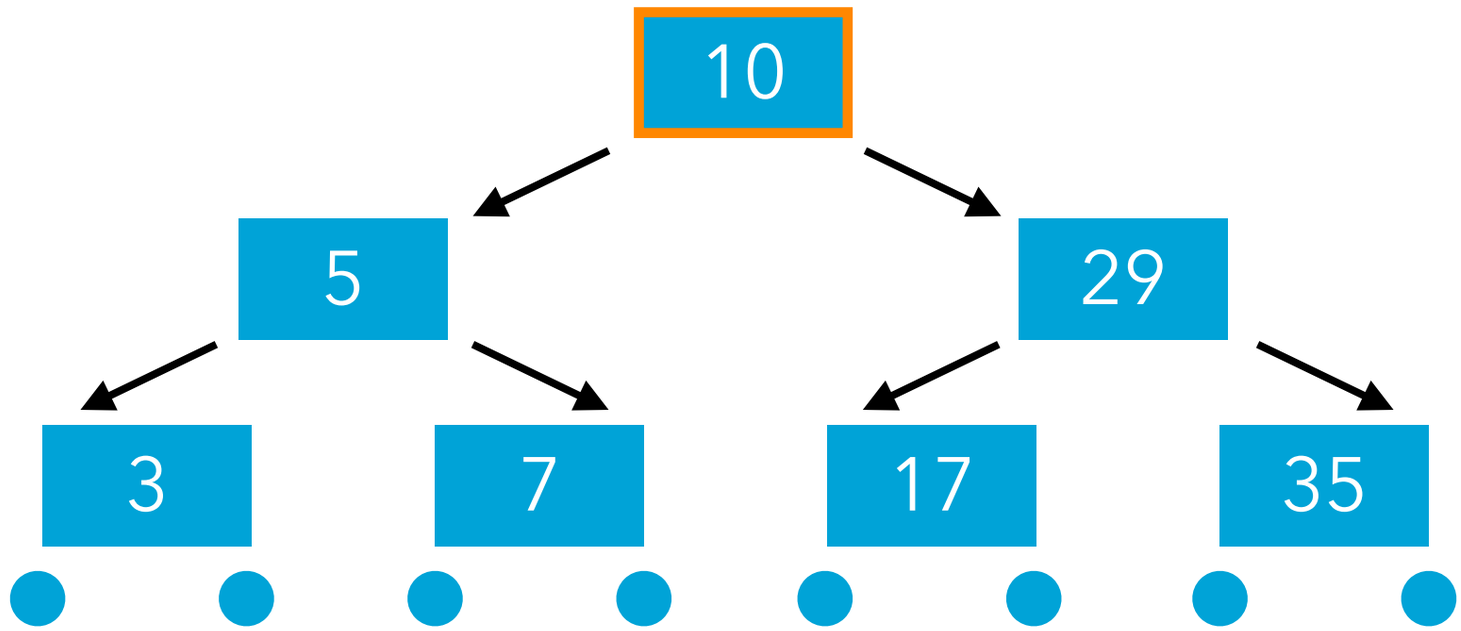
Exemple d'insertion

Exemple : on insère l'entier 42.



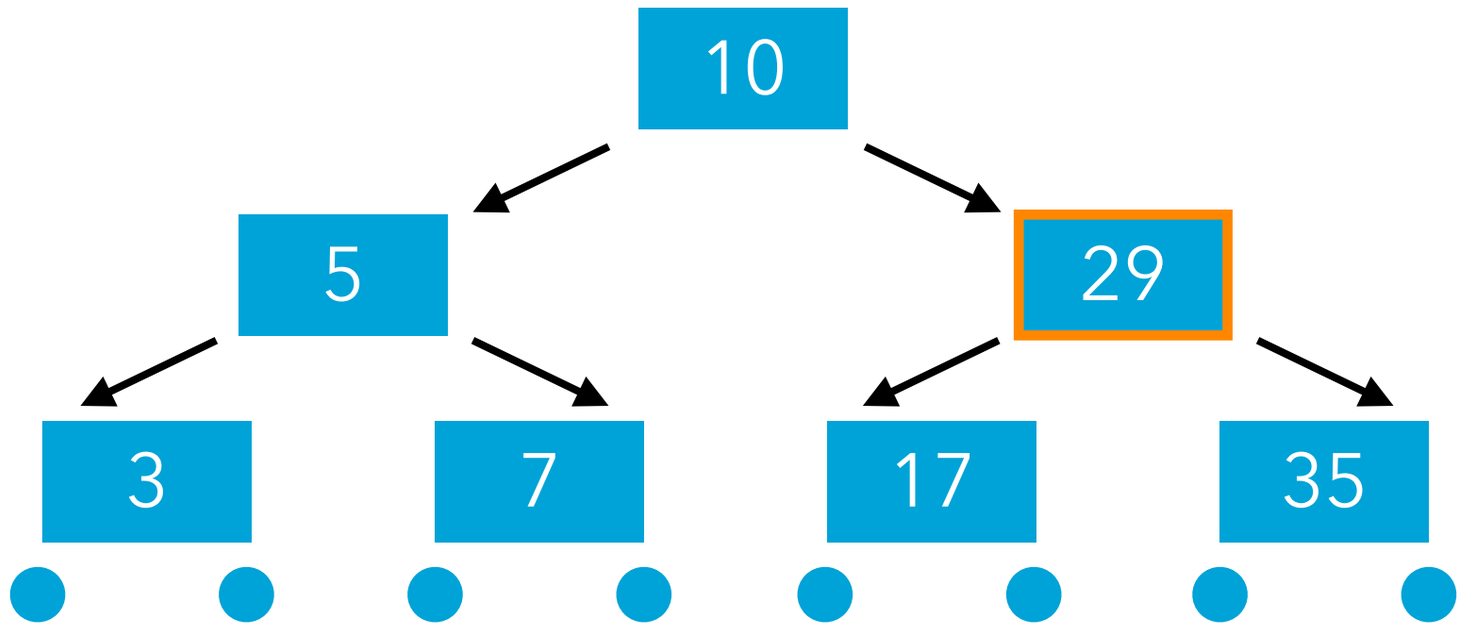
Exemple d'insertion

Exemple : on insère l'entier 42.



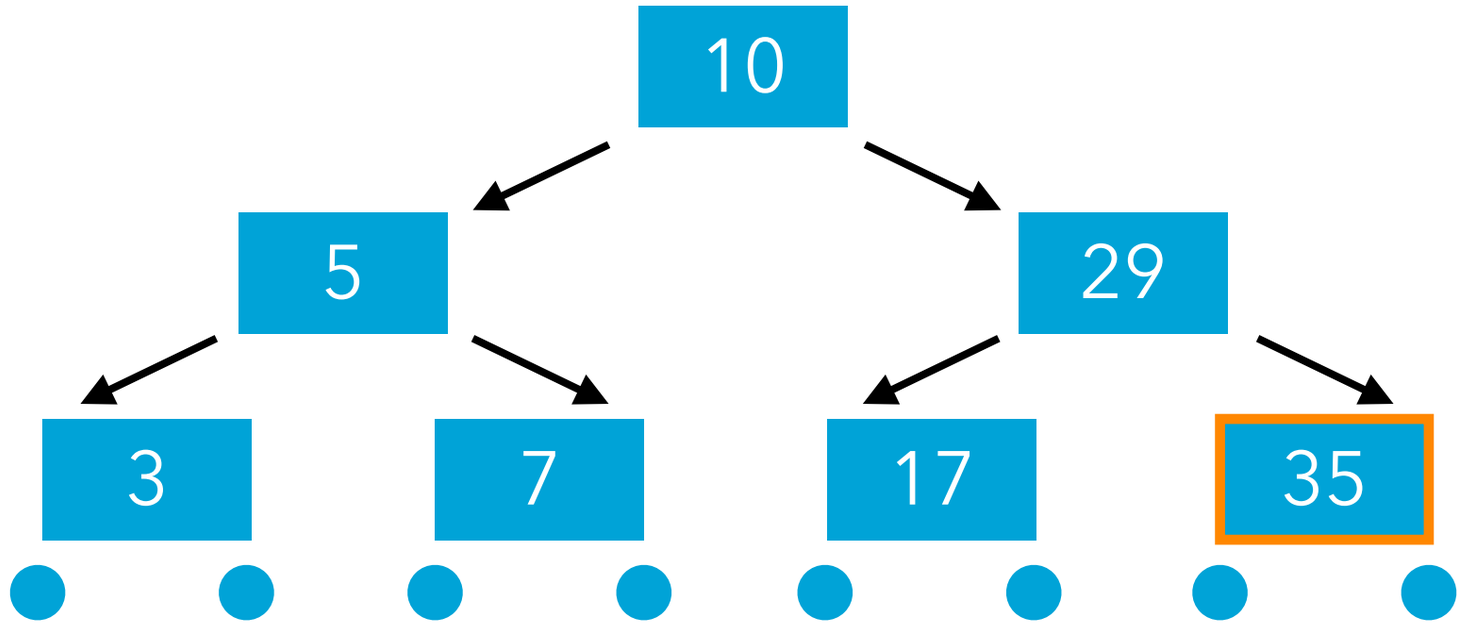
Exemple d'insertion

Exemple : on insère l'entier 42.



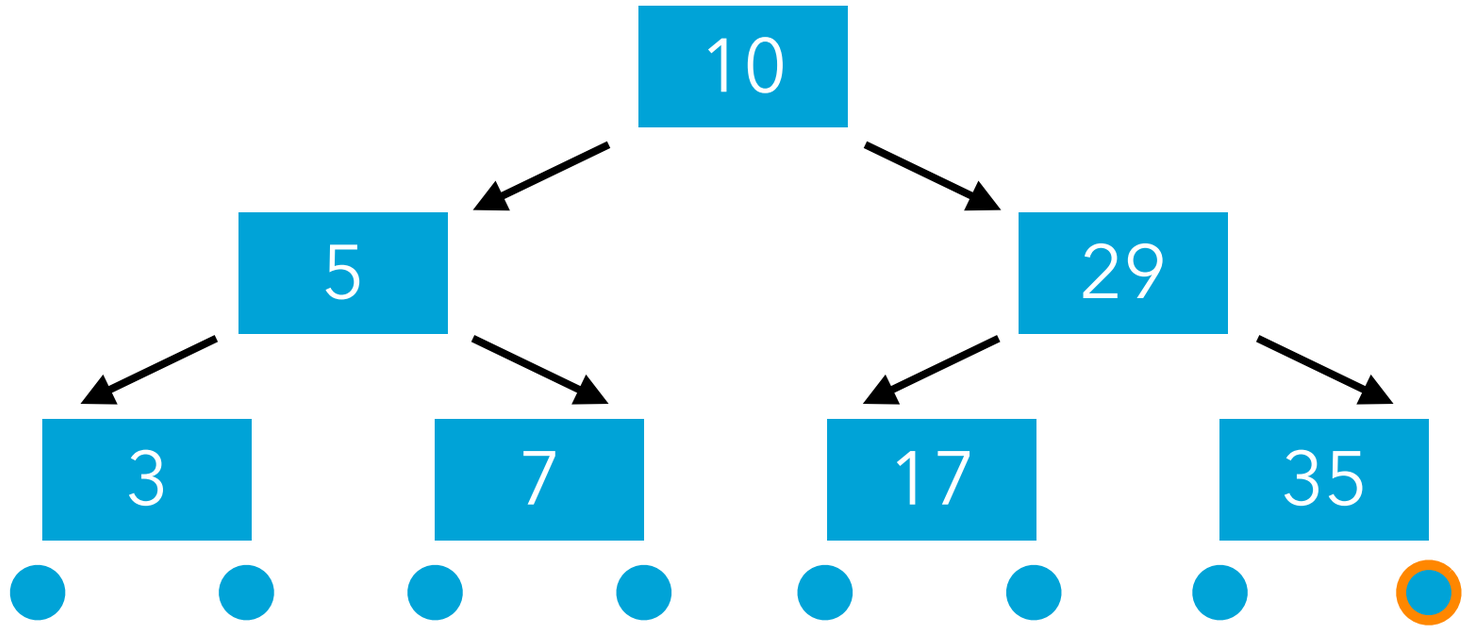
Exemple d'insertion

Exemple : on insère l'entier 42.



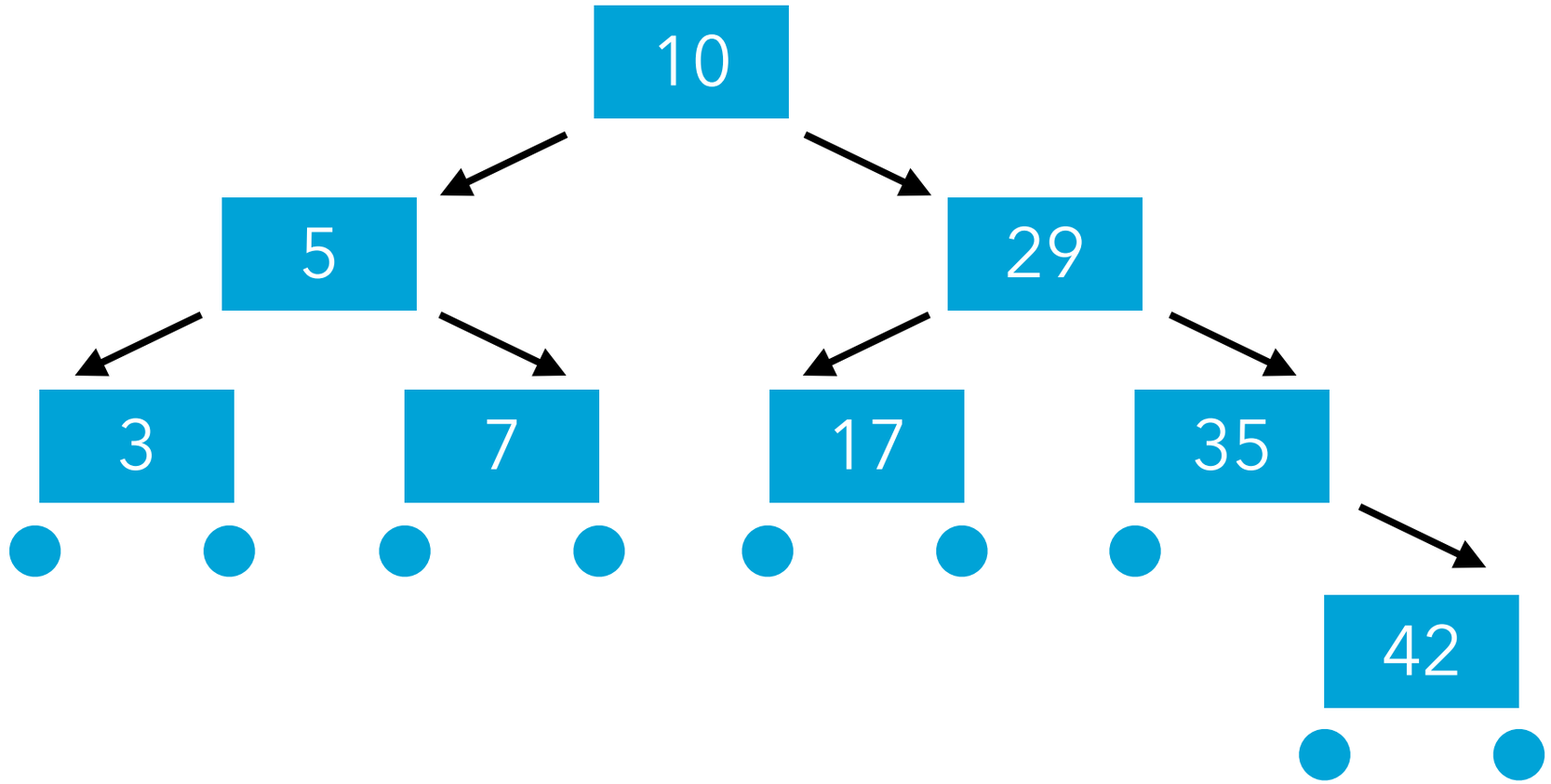
Exemple d'insertion

Exemple : on insère l'entier 42.



Exemple d'insertion

Exemple : on insère l'entier 42.



Code (1)

```
interface Tree {
   …
   public Tree add(int newElem);
}

class Leaf implements Tree {
   public Tree add(int newElem) {
        return new Node(new Leaf(),
                        newElem,
                        new Leaf());
   }
}
```

Code (2)

```
class Node implements Tree {
  …
  public Tree add(int newElem) {
    if (newElem < v) {
      s = s.add(newElem);
    } else if (newElem > v) {
      g = g.add(newElem);
    } else
      ;
    return this;
  }
}
```

élément déjà présent!

Suppression

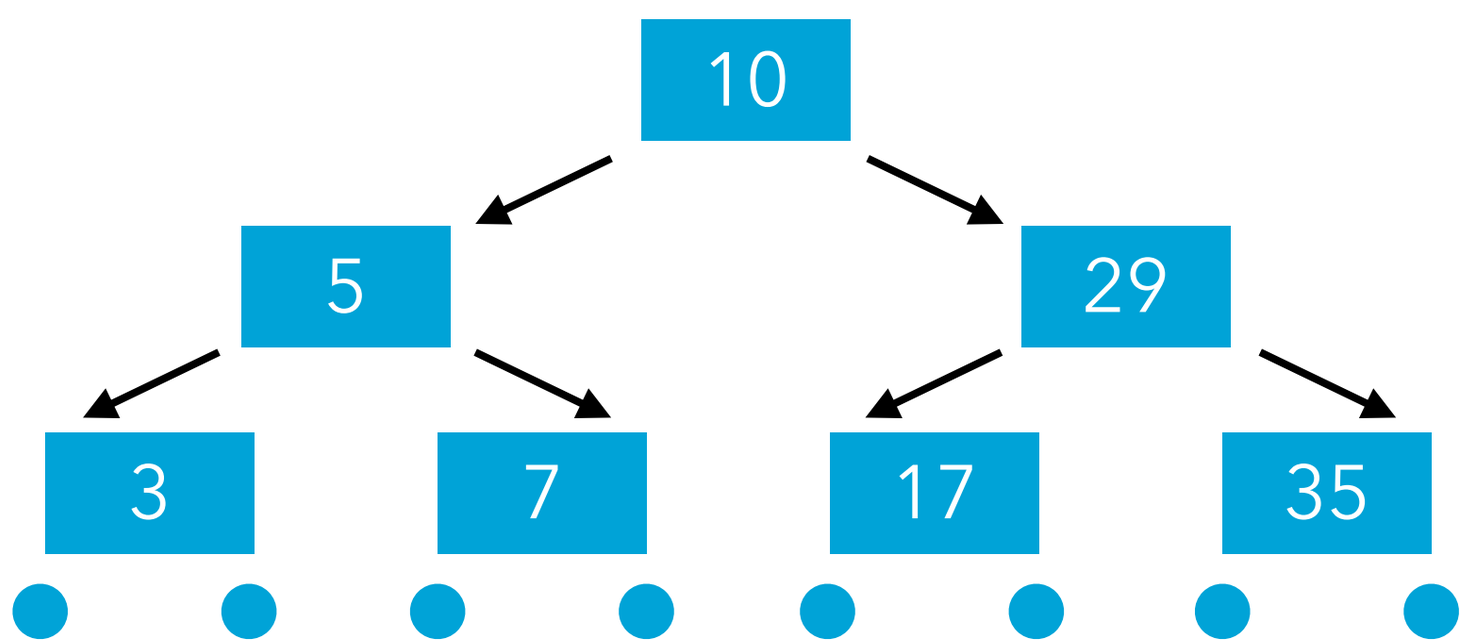
Algorithme de suppression

La suppression d'un élément dans un arbre de recherche est légèrement plus complexe que les autres opérations. On peut distinguer trois cas :

1. le nœud contenant l'élément n'a aucun fils – on le supprime,
2. le nœud contenant l'élément a un fils unique – on le remplace par ce fils,
3. le nœud contenant l'élément a deux fils – cas plus complexe, détaillé plus loin.

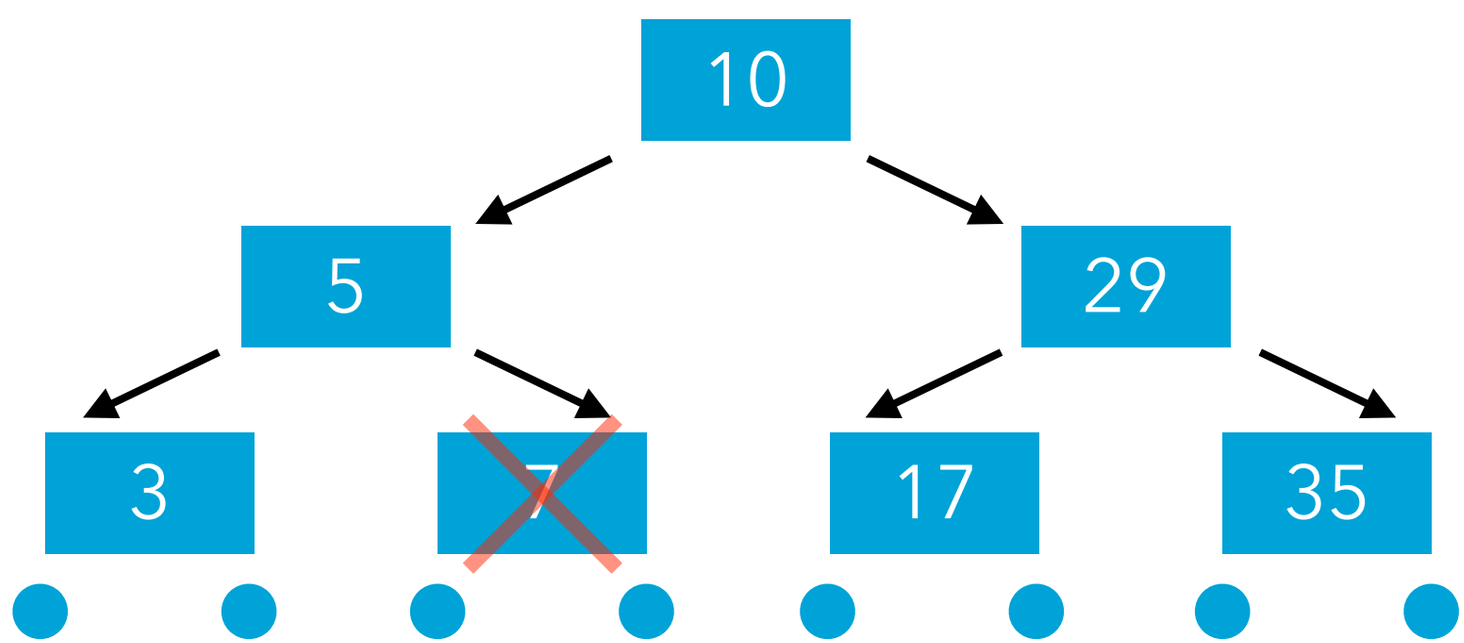
Exemple de suppression

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



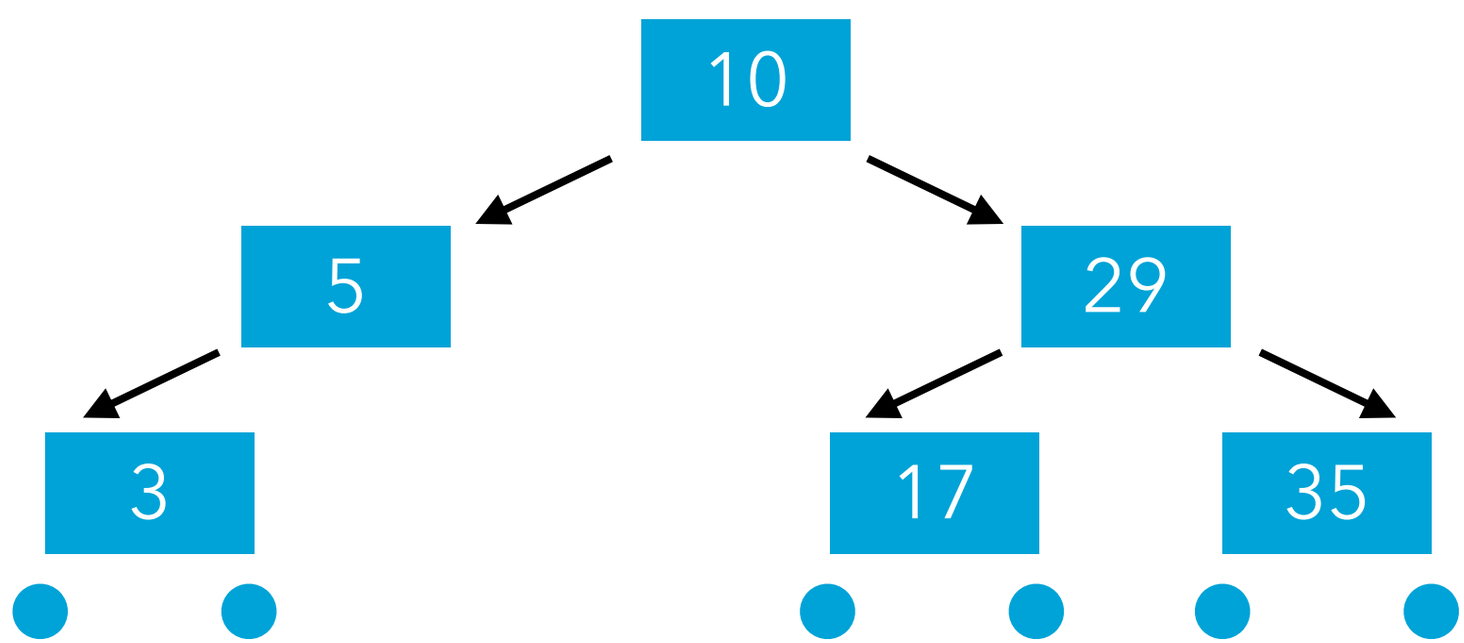
Exemple de suppression

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



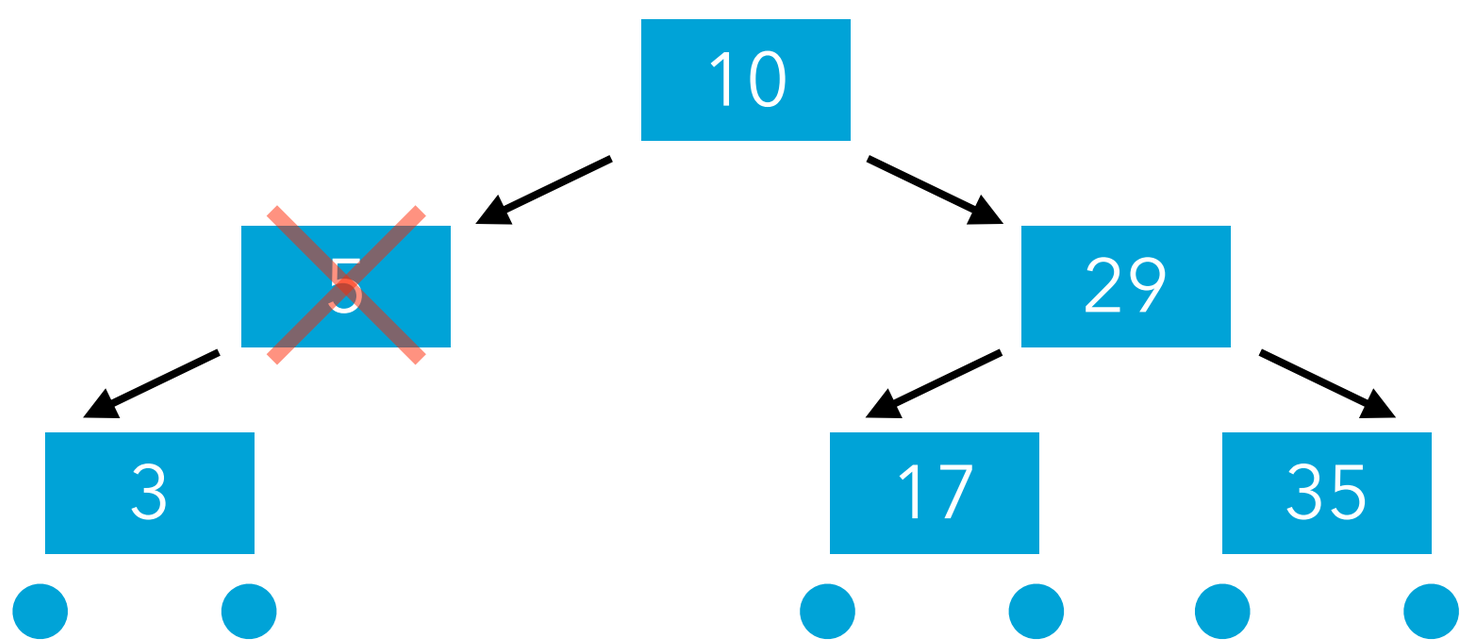
Exemple de suppression

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



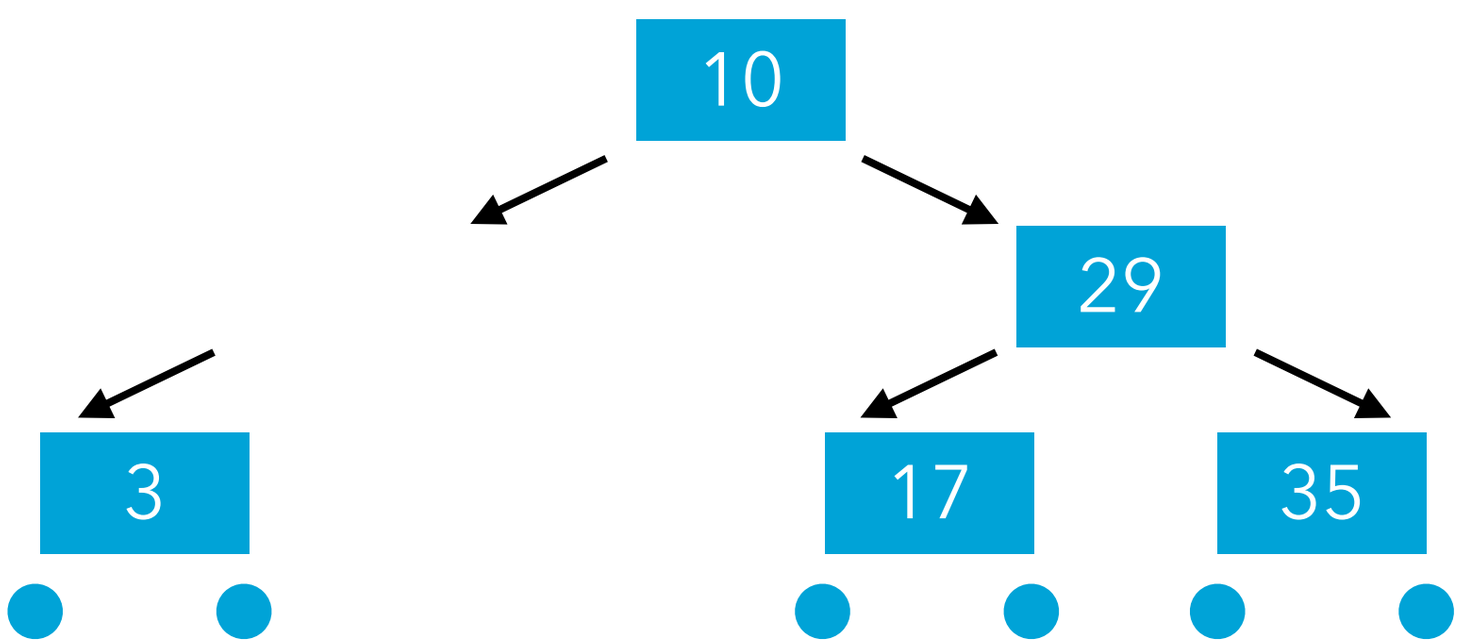
Exemple de suppression

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



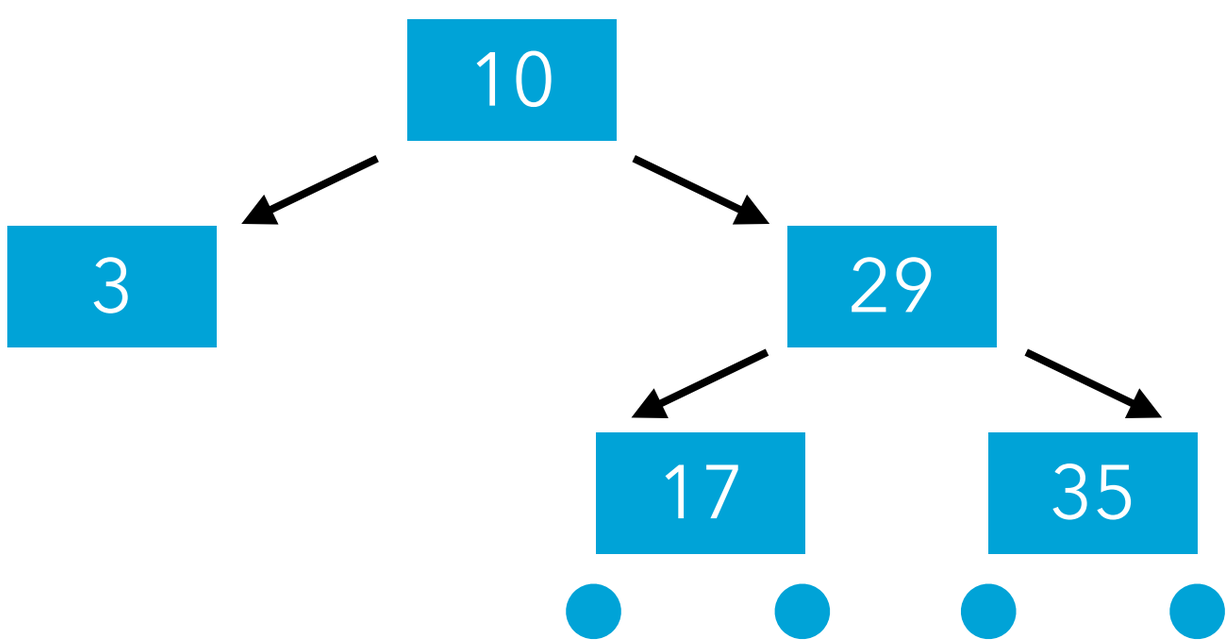
Exemple de suppression

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



Exemple de suppression

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



Suppression : cas complexe

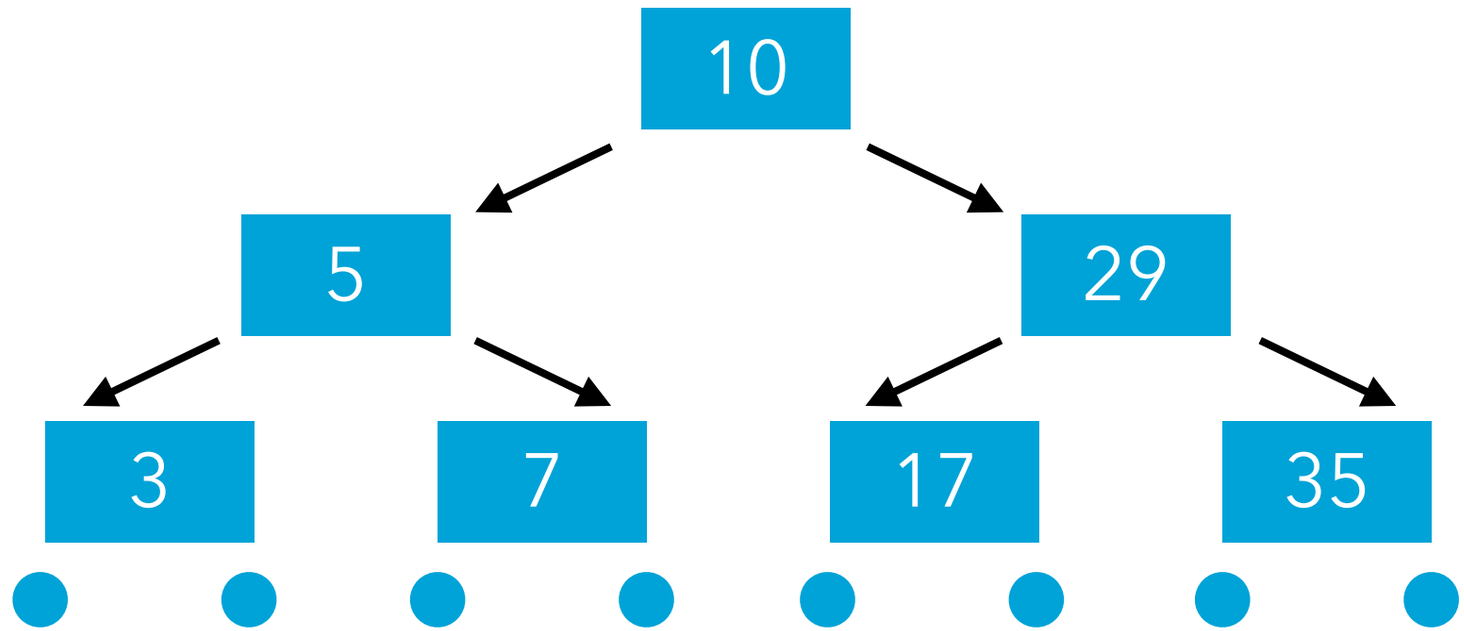
Pour supprimer un élément e stocké dans un nœud n possédant deux fils, l'idée est de trouver dans l'arbre un autre élément qui puisse remplacer e sans violer l'invariant de l'arbre de recherche.

Deux candidats s'imposent rapidement :

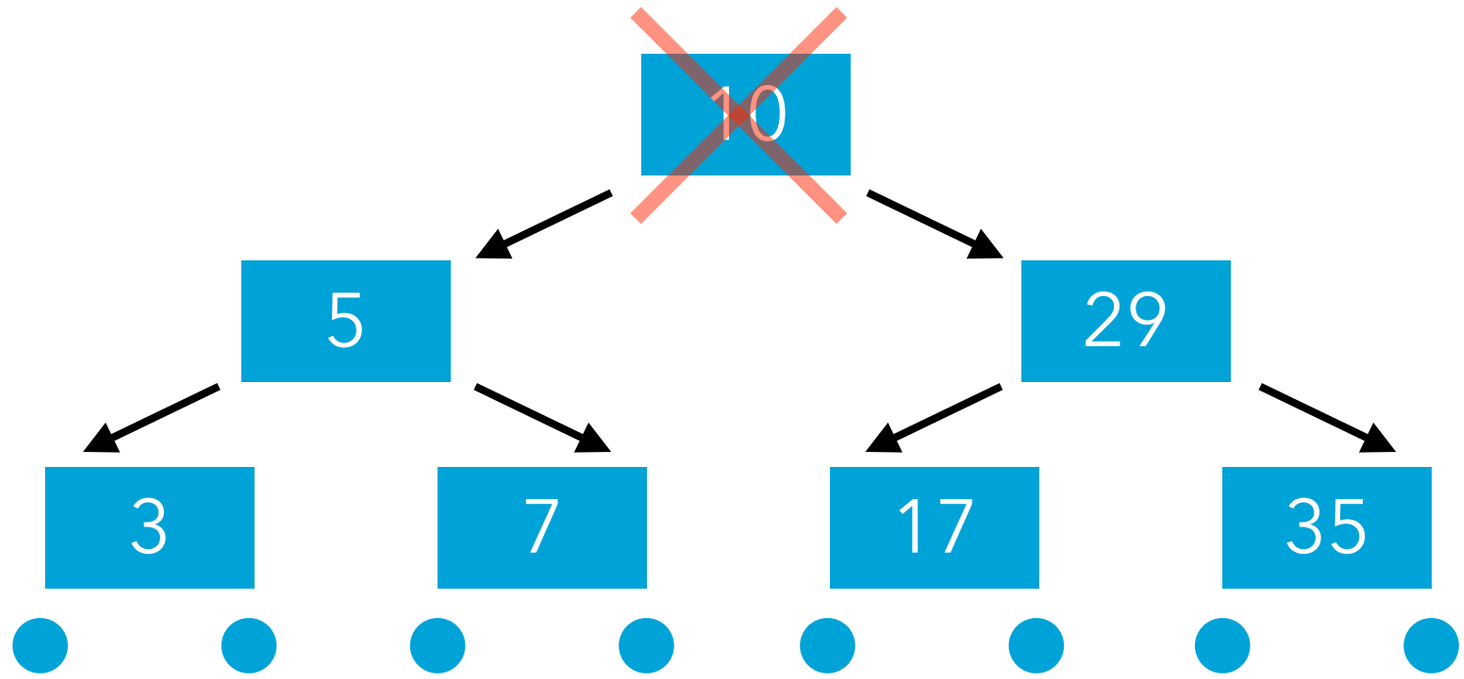
1. l'élément qui précède directement e, c-à-d le plus grand élément du fils gauche de n, ou
2. l'élément qui suit directement e, c-à-d le plus petit élément du fils droite de n.

Etant donnée la structure d'un arbre binaire de recherche, il est certain que les nœuds de ces deux candidats possèdent au plus un fils !

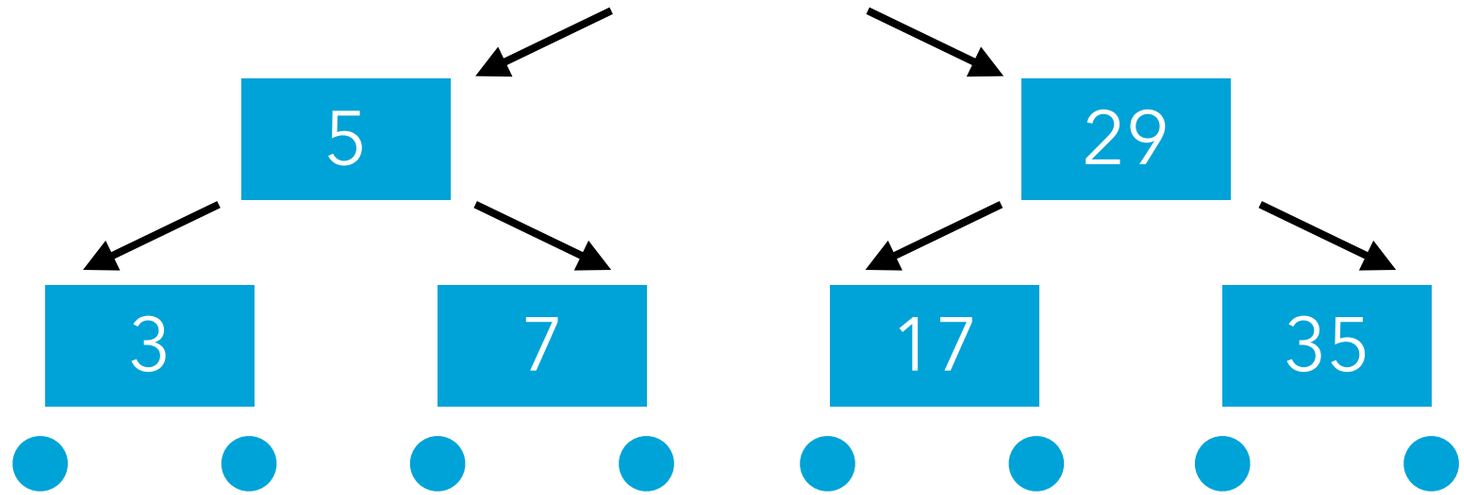
Exemple de suppression



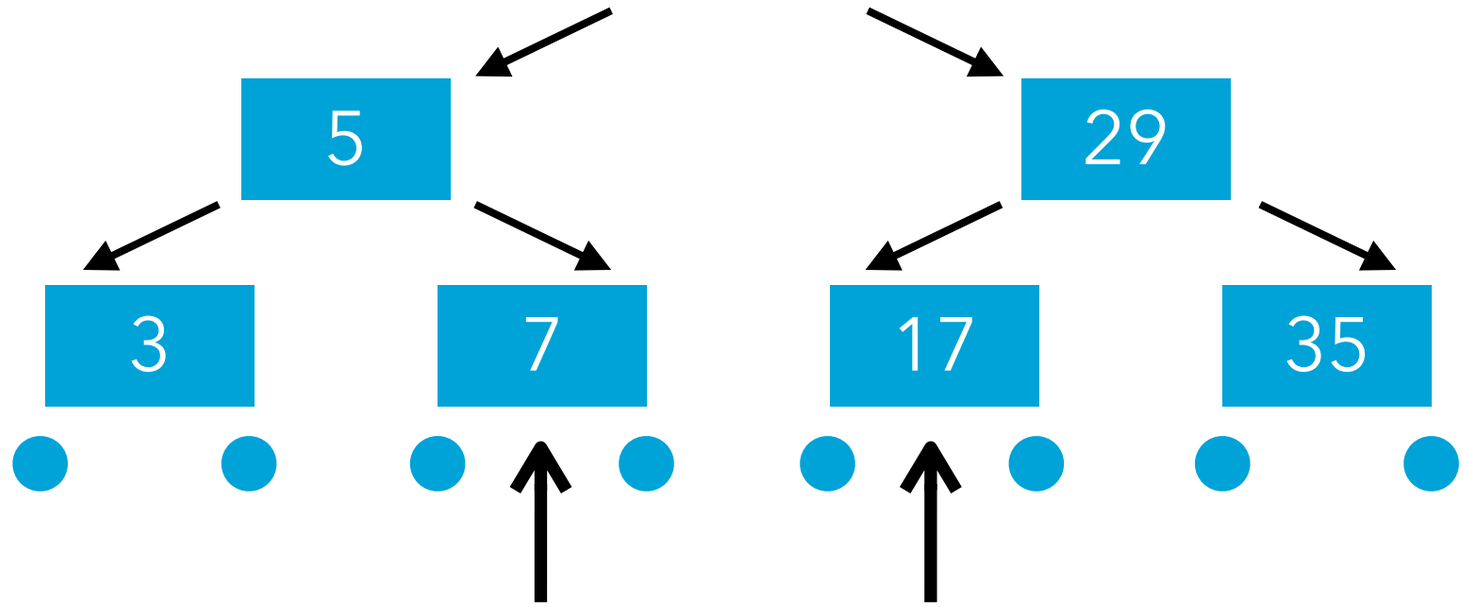
Exemple de suppression



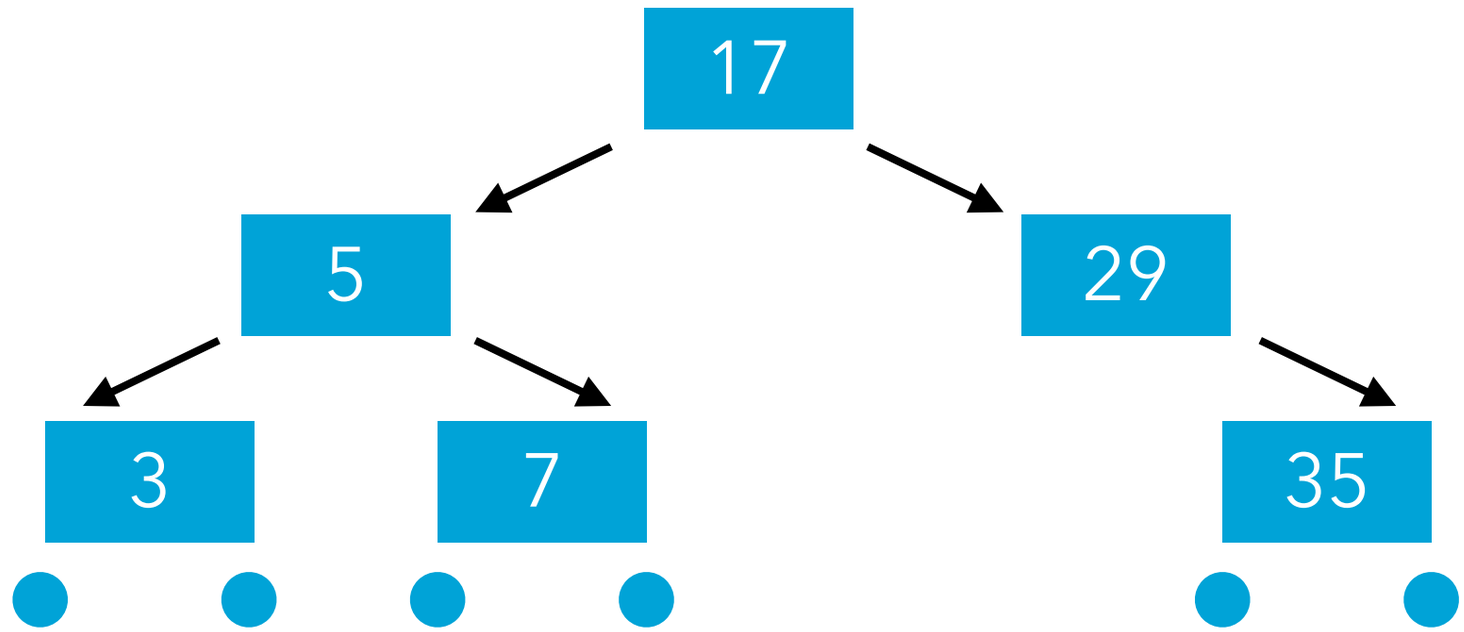
Exemple de suppression



Exemple de suppression



Exemple de suppression



Généralisation du type des éléments

Type des éléments

Jusqu'à présent, nous n'avons parlé que d'arbres de recherche contenant des nombres entiers.

Pour généraliser ces arbres et leur permettre de contenir des éléments de type quelconque, une idée s'impose immédiatement : il faut utiliser la généricité !

Malheureusement, les choses ne sont pas tout à fait aussi simples, comme nous allons le voir.

Arbres génériques

Première tentative de définition d'arbres de recherche génériques :

```
interface Tree<E> {
  public boolean contains(E e);
}
class Leaf<E> implements Tree<E> {
  public boolean contains(E e) {
    return false;
  }
}
```

Arbres générique

```
class Node<E> implements Tree<E> {
  private final Tree<E> s, g;
  private final E v;
  public Node(E v, Tree<E> s, Tree<E> g)
    { this.v = v; this.s = s; this.g = g; }
  public boolean contains(E e) {
    if (e < v) {
      return s.contains(e);
    } else if (e > v) {
      return g.contains(e);
    } else { // e == v
      return true;
    }
  }
}
```

Quel est le problème de cette méthode ?

Relation d'ordre

Le problème de la classe `Node` est qu'elle ne fait pas abstraction de la relation d'ordre sur les éléments stockés dans l'arbre – c-à-d de la manière dont ils doivent être comparés !
Pour faire abstraction de cette relation, on peut penser à deux solutions :

1. on admet que les objets stockés dans les nœuds de l'arbre « savent » se comparer entre eux,
2. on trouve un moyen de représenter la relation d'ordre et on la passe en argument au moment de la création de l'arbre.

Examinons-les l'une après l'autre.

Solution 1 :
ordre interne

Relation d'ordre interne

Si tous les objets stockés dans les nœuds de l'arbre savaient se comparer entre-eux, le problème serait résolu !
Admettons donc que tous les éléments sont équipés d'une méthode `compareTo` telle que:

`x.compareTo(y) < 0` ssi `x < y`
`x.compareTo(y) == 0` ssi `x = y`
`x.compareTo(y) > 0` ssi `x > y`

et voyons comment il est possible de l'utiliser pour écrire une version correcte de la méthode `contains` de la classe `Node`.

Relation d'ordre interne

Au moyen d'une telle méthode `compareTo`, il devient aisé d'écrire la méthode `contains` de la classe `Node` générique :

```
  public boolean contains(E e) {
    int comp = e.compareTo(v);
    if (comp < 0) {
      return s.contains(e);
    } else if (comp > 0) {
      return g.contains(e);
    } else {
      return true;
    }
  }
```

Définition de `compareTo`

Nous avons pour l'instant fait l'hypothèse que les éléments stockés dans l'ensemble possèdent une méthode `compareTo`.

Malheureusement, cette méthode n'est pas définie dans la classe `Object`, qui contient bien une méthode `equals`, mais pas de méthode de comparaison générale.

Comment faire ?

Interface de comparaison

Idée : définir une interface contenant la méthode `compareTo`, et s'assurer ensuite que les éléments de l'ensemble implémentent cette interface.

Premier essai :

```
interface Comparable1 {
  public int compareTo(Object that);
}
```

Borne des éléments

Afin que les éléments de l'ensemble soient comparables, il faut qu'ils implémentent l'interface `Comparable1`. Pour garantir cela, on peut placer une **borne** (supérieure) sur le paramètre de type, au moyen de la syntaxe suivante :

```
interface Tree<E extends Comparable1>
  { … }
// idem pour Leaf, Node et Set
```

Cette borne garantit que seuls des sous-types de `Comparable1` pourront être utilisés pour instancier le type générique `Tree`.
En d'autres terme, on a la garantie que toute valeur de type `E` implémente `Comparable1` (et possède donc la méthode `compareTo`).

Entiers comparables

Essayons maintenant de définir une classe d'entiers comparables, c-à-d implémentant l'interface `Comparable1`.

```
class MyInteger1 implements Comparable1 {
  private final int v;
  public MyInteger1(int v) { this.v = v; }
  public int compareTo(Object that) {
      MyInteger1 thatI = (MyInteger1)that;
      if (v < thatI.v) return -1;
      else if (v > thatI.v) return 1;
      else return 0;
  }
}
```

Le transtypage est gênant, d'autant qu'il peut échouer...

Peut-on s'en débarrasser ?

Interface de comparaison

Problème : notre interface `Comparable1` est un peu trop générale, car la méthode `compareTo` accepte un objet quelconque.
Solution : ajoutons un paramètre de type `T` à l'interface, qui est le type des objets avec lesquels la comparaison est possible. On obtient alors :

```
interface Comparable2<T> {
  public int compareTo(T that);
}
```

Entiers comparables (v2)

Au moyen de l'interface `Comparable2`, on peut écrire une classe d'entiers comparables qui évite tout transtypage gênant :

```
class MyInteger2
    implements Comparable2<MyInteger2> {
  private final int v;
  public MyInteger2(int v) { this.v = v; }
  public int compareTo(MyInteger2 that) {
    if (v < that.v) return -1;
    else if (v > that.v) return 1;
    else return 0;
  }
}
```

`java.lang.Comparable`

La bibliothèque standard Java définit déjà une interface de comparaison :

`java.lang.Comparable<T>`

Elle est définie exactement comme notre `Comparable2`, et est implémentée par une grande quantité de classes standard (`String`, `Date`, ...).

L'ordre défini par la méthode `compareTo` est appelé **l'ordre naturel** d'une classe.

Solution 2 :
ordre externe

Comparateurs

L'autre manière de spécifier la relation d'ordre est de la représenter de manière explicite et de la passer en argument lors de la création de l'ensemble.
Pour représenter cette relation, on utilise très naturellement un objet doté d'une méthode permettant de comparer deux éléments.
On appelle un tel objet un **comparateur**.

Interface des comparateurs

Pour définir l'interface des comparateurs, on utilise une technique similaire à celle utilisée précédemment : l'interface est générique, et son paramètre de type représente le type des objets que le comparateur sait comparer.

```
interface Comparator<T> {
  int compare(T obj1, T obj2);
}
```

La méthode `compare` retourne un entier négatif si elle considère que `obj1` est inférieur à `obj2`, un entier positif si elle considère que `obj1` est supérieur à `obj2`, et zéro sinon.

java.util.Comparator

Une fois encore, la bibliothèque Java contient une définition d'une interface pour les comparateurs, identique à la nôtre :

`java.util.Comparator<T>`

Pour comparer des valeurs, la bibliothèque Java accepte en général d'utiliser soit un comparateur, soit l'ordre naturel des objets, ce qui offre une grande flexibilité.

Ordre interne ou externe ?

Pourquoi avoir deux techniques – l'ordre naturel et les comparateurs – pour représenter la relation d’ordre ?

- L’ordre naturel est plus léger, dans le sens où il n’est pas nécessaire de passer explicitement la relation d’ordre.
- Les comparateurs sont plus flexibles, dans le sens où il est possible d’en définir plusieurs pour un seul type d’éléments – p.ex. pour ordonner des employés soit par nom, soit par âge.

Equilibre des arbres de recherche

Arbres dégénérés

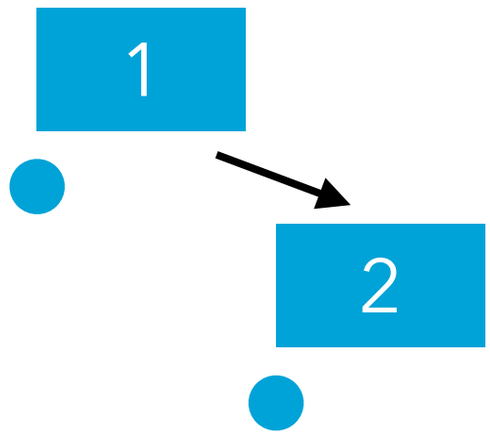
Que se passe-t-il si on insère successivement les entiers 1,2,3,4 dans un arbre de recherche initialement vide ?

Arbres dégénérés

Que se passe-t-il si on insère successivement les entiers 1,2,3,4 dans un arbre de recherche initialement vide ?

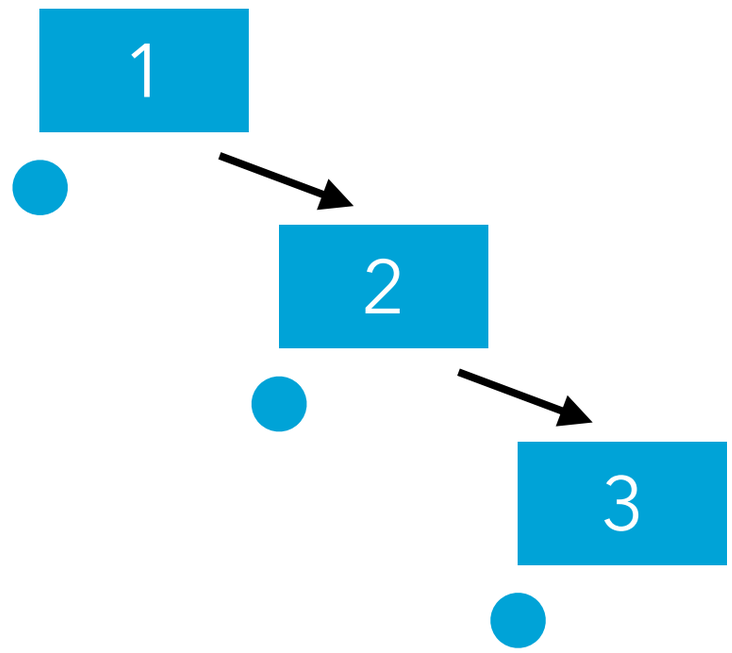
Arbres dégénérés

Que se passe-t-il si on insère successivement les entiers 1,2,3,4 dans un arbre de recherche initialement vide ?



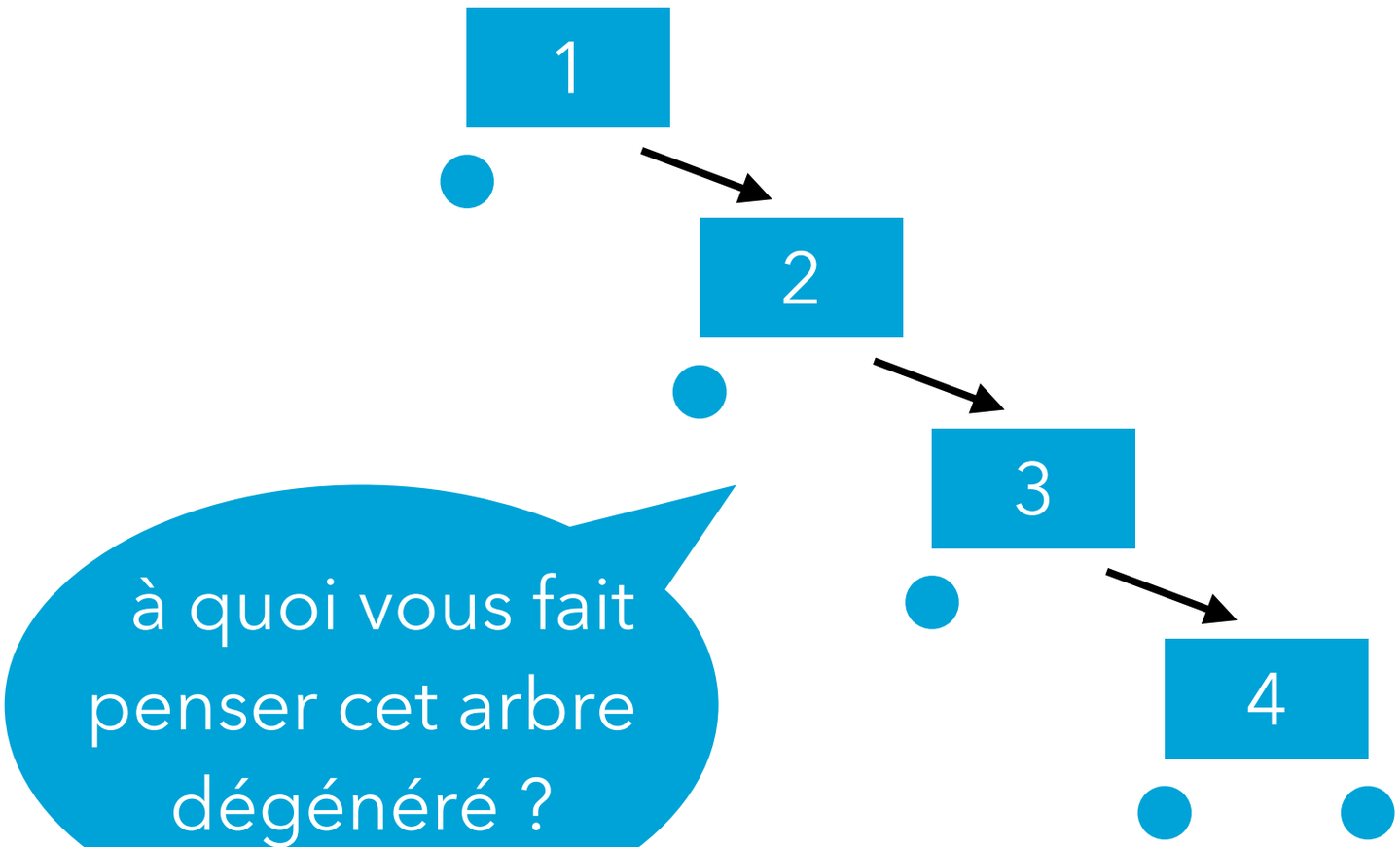
Arbres dégénérés

Que se passe-t-il si on insère successivement les entiers 1,2,3,4 dans un arbre de recherche initialement vide ?



Arbres dégénérés

Que se passe-t-il si on insère successivement les entiers 1,2,3,4 dans un arbre de recherche initialement vide ?



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 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 de taille n a dégénéré en liste, sa hauteur est 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,
- 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.

A.b.r dans l'API Java

API Java

Bonne nouvelle : les arbres binaires de recherche font déjà partie de la bibliothèque standard Java !
La classe `java.util.TreeSet<E>` représente des ensembles au moyen d'arbres rouge-noir.
Pour ordonner les éléments, `TreeSet` peut soit utiliser l'ordre naturel des éléments – qui doivent alors implémenter l'interface `Comparable` – ou un comparateur explicite.

Utilisation de l'ordre naturel

Beaucoup de classes Java implémentent l'interface `Comparable`. On peut ainsi facilement les placer dans un `TreeSet`.

Exemple avec la classe `String` :

```
TreeSet<String> e = new TreeSet<>();
e.add("Jean");
e.add("Marie");
e.contains("Marie");   // true
e.contains("Charles"); // false
```

Utilisation d'un comparateur

Si une classe n'implémente pas l'interface `Comparable`, ou si on désire utiliser un autre ordre que l'ordre naturel, il est possible d'utiliser un comparateur. Exemple :

```
  class Employee {
    public final String name;
    public Employee(String name) { … }
  }
class EmpComp
    implements Comparator<Employee> {
  public int compare(Employee e1,
                     Employee e2) {
    return e1.name.compareTo(e2.name);
  }
}
```

Utilisation d'un comparateur

Pour placer des instances de la classe `Employee` dans un ensemble `TreeSet`, il suffit de passer une instance du comparateur au moment de la création de l'ensemble.

```
TreeSet<Employee> e =
  new TreeSet<Employee>(new EmpComp());
e.add(new Employee("Jean"));
e.add(new Employee("Marie"));
e.contains(new Employee("Marie"));   // t
e.contains(new Employee("Charles")); // f
```

Résumé

La recherche dichotomique permet de trouver un élément dans un tableau trié en O(log n).
Cette technique conduit à l'organisation des éléments d'un ensemble en un arbre binaire de recherche.
Une difficulté rencontrée lors de la mise en œuvre des arbres de recherche génériques est la spécification de la relation d'ordre. Nous avons examiné deux techniques : l'ordre naturel et les comparateurs.