

# Méthodes de conception de programmes

## Devoir 2 : 1, 2, 3... Arbres !

Alexandre GOBEAUX<sup>a</sup>, Louis NAVARRE<sup>a</sup>, Gilles PEIFFER<sup>a</sup>

<sup>a</sup>*École Polytechnique, Université catholique de Louvain, Place de l'Université 1, 1348 Ottignies-Louvain-la-Neuve, Belgique*

### Abstract

Ce papier donne les invariants de représentation, la fonction d'abstraction et les spécifications des fonctions `insert` et `join` pour une implémentation des arbres 2-3 basée sur Sedgewick and Wayne (2011) et Wikipedia contributors (2018)<sup>1</sup>.

### 1. Invariant de représentation

Commençons par définir quelques fonctions auxiliaires :

- `size(t)` : donne le nombre de nœuds d'un arbre  $t$  ;
- `height(t)` : donne la hauteur d'un arbre  $t$  ;
- `type(t)` : donne le nombre de sous-arbres du nœud source de l'arbre  $t$ .

Afin d'alléger la notation de l'invariant de représentation  $\text{ok}_{\text{Arbre 2-3}}(t)$ , voici quelques conventions supplémentaires.

- Si le nœud source de  $t$  est un 2-nœud, alors  $\ell$  et  $r$  dénotent respectivement le sous-arbre de gauche et de droite de  $t$ , alors que  $a$  dénote la donnée de son nœud source.
- Si le nœud source de  $t$  est un 3-nœud, alors  $\ell$ ,  $m$  et  $r$  dénotent respectivement le sous-arbre de gauche, du milieu et de droite de  $t$ , alors que  $a < b$  sont les données du nœud source.

$$f(t) = \left( \text{size}(\ell) > 0 \wedge \text{size}(r) > 0 \right) \wedge \left( \text{height}(\ell) = \text{height}(r) \right) \wedge \left( \forall \lambda \in \ell, \varrho \in r : \lambda < a \leq \varrho \right), \quad (1)$$

$$g(r) = \left( \text{size}(\ell) > 0 \wedge \text{size}(m) > 0 \wedge \text{size}(r) > 0 \right) \wedge \left( \text{height}(\ell) = \text{height}(m) = \text{height}(r) \right) \wedge \left( \forall \lambda \in \ell, \mu \in m, \varrho \in r : \lambda < a \leq \mu < b \leq \varrho \right). \quad (2)$$

L'invariant de représentation est alors donné par

$$\boxed{\text{ok}_{\text{Arbre 2-3}}(t) \equiv \text{size}(t) = 0 \vee \left( \text{type}(t) = 2 \wedge f(t) \right) \vee \left( \text{type}(t) = 3 \wedge g(t) \right).} \quad (3)$$

### 2. Fonction d'abstraction

Définissons une séquence d'entiers  $\langle \cdot \rangle$  par la règle de concaténation suivante :

$$\langle \langle a \rangle, \langle b \rangle \rangle = \langle \text{concat}(a, b) \rangle, \quad (4)$$

où  $a$  et  $b$  sont des séquences potentiellement vides ou des nombres. En utilisant la même notation que pour l'invariant de représentation, La fonction d'abstraction  $\text{abs}_{\text{Arbre 2-3}}(t)$  est alors donnée par

$$\boxed{\text{abs}_{\text{Arbre 2-3}}(t) = \begin{cases} \langle \rangle, & \text{si } t = \text{Vide}, \\ \langle \text{abs}_{\text{Arbre 2-3}}(\ell), a, \text{abs}_{\text{Arbre 2-3}}(r) \rangle, & \text{si } t = \text{Deux}(\ell, a, r), \\ \langle \text{abs}_{\text{Arbre 2-3}}(\ell), a, \text{abs}_{\text{Arbre 2-3}}(m), b, \text{abs}_{\text{Arbre 2-3}}(r) \rangle, & \text{si } t = \text{Trois}(\ell, a, m, b, r). \end{cases}} \quad (5)$$

### 3. Spécifications formelles

3.1. Spécification de `insert(t: Tree, i: int) returns (t': Tree)`

3.1.1. Précondition

La precondition est simplement que l'invariant de représentation soit respecté.

$$\boxed{\text{Pre} : \text{ok}_{\text{Arbre 2-3}}(t).} \quad (6)$$

Email addresses: alexandre.gobeaux@student.uclouvain.be (Alexandre GOBEAUX), navarre.louis@student.uclouvain.be (Louis NAVARRE), gilles.peiffer@student.uclouvain.be (Gilles PEIFFER)

1. La définition des arbres 2-3 de cette dernière source n'était pas valable lorsque l'arbre 2-3 contient des doublons, ce qui a été corrigé.

### 3.1.2. Postcondition

Comme la fonction `insert` doit conserver le rep-invariant, il fait également partie de la postcondition. En plus de cela, on requiert également que le multiensemble des éléments du nouvel arbre soit égal au multiensemble correspondant à l'arbre résultant de l'union de l'arbre original et de  $i$ , où les multiensembles sont définis comme dans Blizard (1991).

$$\boxed{\text{Post} : \text{ok}_{\text{Arbre } 2-3}(t') \wedge \text{elements}(t') = \text{elements}(t) \cup \{\{i\}\}}. \quad (7)$$

Ici, `elements` est un opérateur qui prend en argument un arbre 2-3 et retourne le multiensemble de ses éléments, alors que la notation  $\{\{ \cdot \}\}$  dénote un multiensemble.

## 3.2. Spécification de `join(t1: Tree, t2: Tree) returns (t: Tree)`

### 3.2.1. Précondition

La fonction `join` a comme precondition que les deux arbres pris en argument soient des arbres 2-3 valables.

$$\boxed{\text{Pre} : \text{ok}_{\text{Arbre } 2-3}(t1) \wedge \text{ok}_{\text{Arbre } 2-3}(t2)}. \quad (8)$$

### 3.2.2. Postcondition

Le résultat de la fonction `join` doit également être un arbre 2-3 valable. En plus de cela, on requiert également que le multiensemble des éléments du nouvel arbre soit égal au multiensemble résultant de l'union des deux arbres initiaux.

$$\boxed{\text{Post} : \text{ok}_{\text{Arbre } 2-3}(t) \wedge \text{elements}(t) = \text{elements}(t1) \cup \text{elements}(t2)}. \quad (9)$$

La fonction `elements` est la même que pour `insert`.

## 4. Code

---

```
datatype Tree = Vide
              | Deux(Tree, int, Tree)
              | Trois(Tree, int, Tree, int, Tree)

method insert2(t: Tree, i:int) returns (ret: Tree, isSame: bool)
  // Requires
  // Ensures
{
  match t
  case Vide =>
    ret := Deux(Vide, i, Vide); // empty
    isSame := false;
  case Deux(left, val, right) => // It was a 2-tree
    if (val ≥ i)
    {
      var newLeft, cont := insert2(left, i);
      if (cont)
      {
        ret := Deux(newLeft, val, right);
        isSame := true;
      }
    }
    else
    {
      match newLeft
      case Trois(leftC, valL, middleC, valR, rightC) =>
        ret := Deux(newLeft, val, right);
        isSame := false;
      case Vide =>
        ret := Deux(newLeft, val, right);
        isSame := true;
      case Deux(leftC, valC, rightC) =>
        // Process to make a three of it
        ret := Trois(leftC, valC, rightC, val, right);
        isSame := true;
    }
  }
```

```

    }
  }
  else if (val < i)
  {
    var newRight, cont := insert2(right, i);
    if (cont)
    {
      ret := Deux(left, val, newRight);
      isSame := true;
    }
    else
    {
      match newRight
      case Trois(leftC, valL, middleC, valR, rightC) ⇒
        ret := Deux(left, val, newRight);
        isSame := false;
      case Vide ⇒
        ret := Deux(left, val, newRight);
        isSame := true;
      case Deux(leftC, valC, rightC) ⇒
        ret := Trois(left, val, leftC, valC, rightC);
        isSame := true;
    }
  }
  else
  {
    ret := t;
    isSame := true;
  }

case Trois(left, valL, middle, valR, right) ⇒
  if (i < valL)
  {
    var newLeft, cont := insert2(left, i);
    if (cont)
    {
      ret := Trois(newLeft, valL, middle, valR, right);
      isSame := true;
    }
    else
    {
      match newLeft
      case Vide ⇒
        ret := Trois(newLeft, valL, middle, valR, right);
        isSame := true;
      case Trois(leftC, valLC, middleC, valRC, rightC) ⇒
        ret := Trois(newLeft, valL, middle, valR, right);
        isSame := true;
      case Deux(leftC, valC, rightC) ⇒
        var rightBranch := Deux(middle, valR, right);
        ret := Deux(newLeft, valL, rightBranch);
        isSame := false;
    }
  }
  else if (i ≥ valR)
  {
    var newRight, cont := insert2(right, i);
    if (cont)
    {
      ret := Trois(left, valL, middle, valR, newRight);
      isSame := true;
    }
    else

```

```

    {
        match newRight
        case Vide =>
            ret := Trois(left, valL, middle, valR, newRight);
            isSame := true;
        case Trois(leftC, valLC, middleC, valRC, rightC) =>
            ret := Trois(left, valL, middle, valR, newRight);
            isSame := true;
        case Deux(leftC, valC, rightC) =>
            var leftBranch := Deux(left, valL, middle);
            ret := Deux(leftBranch, valR, newRight);
            isSame := false;
    }
}
else if (valL < i & i < valR)
{
    var newMiddle, cont := insert2(middle, i);
    if (cont)
    {
        ret := Trois(left, valL, newMiddle, valR, right);
        isSame := true;
    }
    else
    {
        match newMiddle
        case Vide =>
            ret := t; // Should be something different but I don't know
            isSame := true;
        case Trois(leftC, valLC, middleC, valRC, rightC) =>
            ret := Trois(left, valL, newMiddle, valR, right);
            isSame := true;
        case Deux(leftC, valC, rightC) =>
            var leftBranch := Deux(left, valL, leftC);
            var rightBranch := Deux(rightC, valR, right);
            ret := Deux(leftBranch, valC, rightBranch);
            isSame := false;
    }
}
else
{
    ret := t;
    isSame := true;
}
}

method insert(t: Tree, i: int) returns (t': Tree)
// Requires ...
// Ensures ...
{
    var newNode, cont := insert2(t, i);
    t' := newNode;
}

method Main()
{
    var t := insert(Vide, 12); print t, "\n";
    t := insert(t, 5); print t, "\n\n";
    t := insert(t, 10); print t, "\n\n";
    t := insert(t, 3); print t, "\n\n";
    t := insert(t, 13); print t, "\n\n";
    t := insert(t, 14); print t, "\n\n";
    t := insert(t, 15); print t, "\n\n";
}

```

```
t := insert(t, 17); print t, "\n\n";  
t := insert(t, 18); print t, "\n\n";  
t := insert(t, 15); print t, "\n\n";  
}
```

---

## Références

Blizard, W. D., 1991. The development of multiset theory. *Mod. Log.* 1 (4), 319–352.

URL <https://projecteuclid.org:443/euclid.rml/1204834739>

Sedgewick, R., Wayne, K., 2011. *Algorithms*, 4th Edition. Addison-Wesley, 21415 Network Place, Chicago, IL 60673, United States.

Wikipedia contributors, 2018. 2–3 tree — Wikipedia, the free encyclopedia. [https://en.wikipedia.org/w/index.php?title=2%E2%80%933\\_tree&oldid=857850249](https://en.wikipedia.org/w/index.php?title=2%E2%80%933_tree&oldid=857850249), [Online; accessed 20-April-2019].