

TP de Java n°3

Un petit interpréteur

MOOB

2025-2026

L'objectif de ce TP est de vous faire travailler sur l'héritage et les interfaces, et de vous faire découvrir les principes permettant d'écrire un petit interpréteur, c'est-à-dire un programme capable d'exécuter d'autres programmes.

Écrire un interpréteur complet capable de lire le code source d'un programme et de l'interpréter est plus complexe que le cadre d'un simple TP. Nous allons simplement nous concentrer sur la partie interprétation à partir d'un arbre de syntaxe abstrait. Un arbre de syntaxe abstrait est une représentation sous forme d'arbre d'un programme. Dans notre cas, un arbre est constitué de noeuds qui peuvent correspondre à des opérations (addition, soustraction etc), des variables ou des scalaires (des entiers).

Par exemple, l'arbre représenté ci-dessous correspond à l'expression `x = x + 42` en Java. La racine de l'arbre est une opération d'affectation qui affecte la partie droite de l'arbre (l'expression qui ajoute 42 à la variable x) à la partie gauche (la variable x).

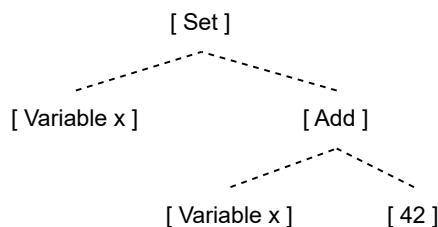


FIGURE 1 – Exemple d'arbre représentant l'expression `x = x + 42`

Dans l'exercice, nous aurons besoin de représenter des arbres sous une forme textuelle. Pour cela, nous affichons un arbre sous une forme préfixée et parenthésée : chaque noeud de l'arbre qui représente une opération est affiché sous la forme `(op e1 ... en)`, où `op` est l'opération, et `e1 ... en` les paramètres. Typiquement, nous afficherons l'arbre ci-dessus sous la forme `(set! x (+ x 42))` : `set!` représente une opération d'affectation, le premier `x` l'élément de gauche, et `(+ x 42)` l'élément de droite (puisque l'opération est une addition avec comme paramètres `x` et `42`). Certains connaisseurs remarqueront que cette écriture est celle du langage de programmation fonctionnelle Scheme.

Méthodes et champs de classe

Dans ce TP, il n'y a pas besoin d'utiliser ni d'attributs de classe ni de méthodes de classe (sauf pour la méthode `main` qui est nécessairement de classe). En d'autres termes, on vous demande de ne pas utiliser le mot clé `static`.

Niveaux d'encapsulation

On vous demande de rendre privés tous les champs et publics tous les constructeurs et toutes les méthodes de vos classes. De façon générale, rendre systématiquement les champs privés est une bonne manière de programmer car vous éviterez ainsi d'utiliser des spécificités internes d'un objet. Pour les méthodes, on peut les définir privées, leur laisser la visibilité par défaut (celle du package) ou les définir publiques, tout dépend du contexte. Ici, la visibilité par défaut est suffisante.

Lisez attentivement le sujet une fois en entier avant de vous lancer dans la programmation.

Exercice 1 : Un simple additionneur

Nous commençons par mettre en œuvre un simple additionneur, c'est-à-dire une classe capable d'additionner deux scalaires.

Question 1.a Dans un projet nommé `ast`, créez une classe `ensiie.ast.Main` contenant une méthode `main`.

Question 1.b Écrivez une classe nommée `Scalar` avec un unique champ (privé) de type entier nommé `value`. Outre un constructeur permettant d'initialiser le champ `value`, ajoutez les méthodes d'instance suivantes à `Scalar` :

- `public int get()` : cette méthode renvoie la valeur du scalaire,
- `public String toString()` : cette méthode retourne une chaîne de caractères représentant la valeur du scalaire en utilisant la méthode statique `Integer.toString(int i)`.

Dans la méthode `main`, créez le scalaire 1 et affichez-le dans le terminal.

Question 1.c On souhaite maintenant créer un arbre capable d'additionner deux scalaires. Écrivez une classe nommée `Add` contenant deux champs de type `Scalar` nommés `left` et `right`. Le constructeur de `Add` doit prendre en argument les opérandes de gauche et de droite et les affecter aux champs. Ajoutez les méthodes suivantes à `Add` :

- `public int execute()` : renvoie le résultat de l'addition de l'opérande de gauche avec l'opérande de droite.
- `public String toString()` : renvoie la chaîne de caractères `(+ e1 e2)`, dans laquelle `e1` est la représentation de `left` sous la forme d'une chaîne de caractères et `e2` celle de `right`.

Dans la méthode `main`, utilisez `Add` pour calculer et afficher le résultat de l'addition de 1 et 2. Vous veillerez aussi à afficher l'expression et à vérifier que l'affichage produit est bien `(+ 1 2)`.

Exercice 2 : Additions imbriquées

Nous souhaitons maintenant rendre notre additionneur générique dans le sens où les opérandes de gauche et de droite doivent pouvoir être des scalaires ou elles-mêmes des opérations d'addition. De cette façon, nous pourrons évaluer une expression comme `(+ (+ 1 2) (+ 3 4))` qui devrait donner comme résultat 10. Pour cela, nous créons une interface générique nommée `Node` représentant un nœud quelconque de l'arbre, c'est-à-dire un scalaire ou un additionneur. La figure ci-dessous présente l'arbre d'héritage/mise en œuvre que nous utilisons :

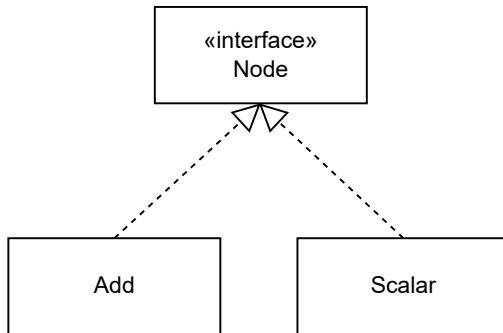


FIGURE 2 – Illustration de l'arbre d'héritage que nous allons mettre en œuvre.

Techniquement, l'interface `Node` doit définir une unique méthode nommée `int execute()` qui va permettre d'évaluer un nœud. Si le nœud est un scalaire, le résultat de son évaluation est sa valeur. Si le nœud est un additionneur, le résultat de son évaluation est la somme du résultat de l'évaluation du membre de gauche avec celle du membre de droite.

Notation `@Override`

L'annotation `@Override` indique au compilateur qu'une méthode donnée est la redéfinition d'une méthode d'une superclasse (interface comprise). En plus de rappeler au programmeur qu'il s'agit de la redéfinition d'une méthode (potentiellement abstraite), la présence de l'annotation `@Override` va forcer le compilateur à vérifier que la méthode a été redéfinie correctement. Cela permet de détecter des erreurs sémantiques dans vos programmes.

Prenez l'habitude d'utiliser cette annotation avant la redéfinition d'une telle méthode !

Question 2.a Créez une interface nommée `Node` définissant la méthode `int execute()`. Comme `Add` possède déjà une méthode `execute`, vous pouvez aussi directement déclarer que `Add` met en œuvre `Node` sans autre modification.

Question 2.b Modifiez `Scalar` de façon à ce que `Scalar` mette en œuvre `Node`. L'évaluation d'un scalaire doit simplement renvoyer sa valeur.

Question 2.c Dans `Add`, modifiez les types des membres de gauche et de droite de façon à ce que ce qu'ils aient le type générique `Node`. Modifiez le constructeur de façon à prendre en paramètre deux nœuds, et modifiez la méthode `execute` de façon à ce qu'elle additionne le résultat de `execute()` sur le membre de gauche avec le résultat de `execute()` sur le membre de droite. Vérifiez que vous arrivez toujours à calculer la somme de 1 et 2.

Question 2.d Construisez l'expression correspondant à `(+ (+ 1 2) (+ 3 4))` dans la méthode `main`. Affichez cette expression sur le terminal avant d'afficher le résultat de son évaluation. Sur le terminal, vous devriez voir :

```
(+ (+ 1 2) (+ 3 4))  
10
```

Exercice 3 : Calculette multi-opérations

Nous souhaitons maintenant ajouter de nouvelles opérations comme la multiplication de deux entiers ou l'inversion du signe d'un entier. Une première approche simple permettant d'ajouter ces opérations serait de créer des classes sur le modèle de `Add` mettant en œuvre `Node`, et effectuant l'opération. Toutefois, comme beaucoup de code serait alors redondant entre ces classes et la classe `Add`, nous préférons utiliser l'héritage pour factoriser au maximum le code. Tout au long de cet exercice, l'affichage produit par votre programme ne devrait pas varier par rapport à l'affichage que vous avez eu à la question précédente.

Question 3.a Nous modifions notre code étape par étape pour introduire une nouvelle classe générique représentant n'importe quelle opération. Dans un premier temps, nous insérons une classe abstraite nommée `Operation` dans l'arbre d'héritage. Cette classe représente une opération quelconque. Pour le moment, cette classe est vide, le but de la question n'étant que de l'insérer de façon à obtenir l'arbre d'héritage/mise en œuvre suivant :

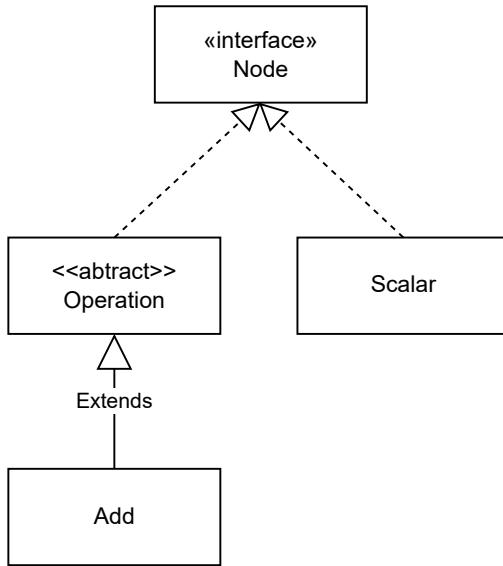


FIGURE 3 – Illustration de l'arbre d'héritage que nous allons mettre en œuvre.

Créez donc une classe abstraite nommée `Operation` et mettant en œuvre `Node`. Cette classe est abstraite car elle ne met pas elle-même en œuvre la méthode `execute` qui est mise en œuvre dans les classes filles. Modifiez ensuite la classe `Add` de façon à ce qu'elle ne mette plus en œuvre `Node`, mais hérite de `Operation`.

Question 3.b La classe `Operation` représente n'importe quelle opération et le nombre d'opérandes n'est donc pas fixe (l'inversion du signe a une unique opérande alors que l'addition en a deux). Pour cette raison, ajoutez à la classe `Operation` :

- un champ privé `Node[] ops` contenant l'ensemble des opérandes,
 - une méthode publique `Node op(int i)` permettant d'accéder à la *i*-ème opérande,
 - une méthode publique `int nbOps()` renvoyant le nombre d'opérandes,
 - un constructeur public `Operation(Node... ops)` permettant d'initialiser le champ `ops` de `Operation`.
- On vous rappelle que `Node... ops` permet de déclarer une méthode avec un nombre variable d'arguments, et que les arguments sont rangés sous la forme d'un tableau dans `ops`.

Pour finir, dans le constructeur de `Add`, utilisez `super` pour initialiser de façon adéquate le tableau d'opérandes de `Operation` via le constructeur de la classe `Operation`. À cette étape, les opérandes sont donc représentées deux fois : une fois dans `Operation` sous la forme d'un tableau, et une fois dans `Add` sous la forme des champs `left` et `right`. Pour le moment, conservez cette double représentation.

Question 3.c Modifiez les méthodes `int execute()` et `String toString()` de `Add` de façon à utiliser la méthode `Node op(int i)` pour accéder aux opérandes. Vous pouvez maintenant vous débarrasser des champs `left` et `right` de `Add` de façon à ne conserver qu'une unique représentation des opérandes dans la classe `Operation`.

Question 3.d Nous souhaitons maintenant rendre la méthode `String toString()` générique en la déplaçant de `Add` vers `Operation`. Dans `Operation`, on connaît les opérandes, mais pas encore l'opération. Pour cette

raison, nous effectuons une étape intermédiaire avant de déplacer la méthode `String toString()`. À cette étape, on vous demande donc juste d'ajouter une méthode abstraite `String opString()` à la classe `Operation` et de la mettre en œuvre dans `Add`. Cette mise en œuvre doit renvoyer la chaîne de caractères "+" . Enfin, modifiez `Add.toString()` de façon à utiliser `opString()` pour générer le +.

Question 3.e Déplacez maintenant `String toString()` de `Add` vers `Operation`. Au lieu de considérer que votre opération a forcément deux opérandes, modifiez la méthode `toString()` pour qu'elle concatène les `nbOps()` opérandes.

Question 3.f Écrivez une classe `Neg` héritant de `Operation` et permettant d'inverser le signe de son opérande. Modifiez la méthode main de façon à afficher et évaluer l'arbre `(+ (+ 1 2) (- (+ 3 4)))` qui devrait donner comme résultat `-4`.

À cette étape, l'arbre d'héritage que vous devriez avoir est le suivant :

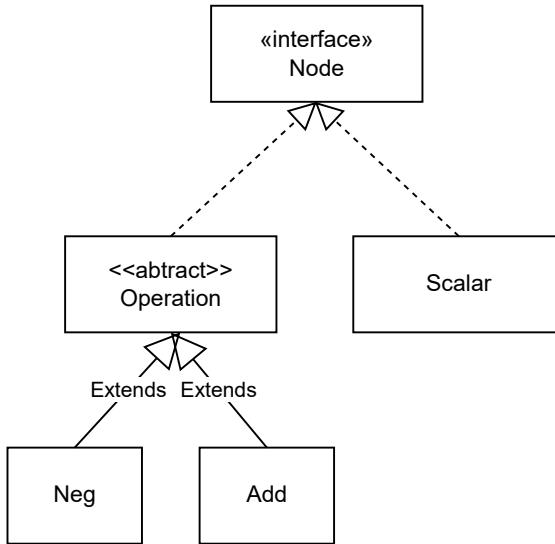


FIGURE 4 – Illustration de l'arbre d'héritage mis en œuvre à la fin de l'exercice.

Exercice 4 : Les variables

Nous souhaitons maintenant ajouter des variables à notre langage. Vous remarquerez qu'une variable est simplement un scalaire possédant un nom. Pour cette raison, nous réutilisons la classe `Scalar` pour mettre en œuvre les variables.

Question 4.a Ajoutez une classe `Variable` héritant de `Scalar` et contenant :

- un champ `String name` indiquant le nom de la variable,
- un constructeur prenant en argument le nom de la variable et pré-initialisant la variable à 0 en utilisant `super`.

Question 4.b Évaluer une variable revient à renvoyer sa valeur. La méthode `int execute()` mise en œuvre dans `Scalar` effectue déjà ce travail et il n'est donc pas nécessaire de la redéfinir dans `Variable`. En revanche, lorsqu'on affiche une variable avec `String toString()`, on veut afficher le nom de la variable et non sa valeur. Redéfinissez donc la méthode `String toString()` dans `Variable` de façon à afficher le nom de la variable. Dans la méthode `main`, allouez une variable `x`, puis affichez et évaluez l'arbre `(+ x 20)`, qui devrait donc être évalué à 20 puisque `x` est pré-initialisé à 0.

Question 4.c Ajoutez une classe `Set`, héritant de `Operation`, et permettant de modifier une variable. Cette opération prend deux arguments : une variable et un noeud quelconque. L'affichage d'une expression doit produire la chaîne de caractère `(set! var val)` où `var` est la variable et `val` un noeud quelconque.

Comme le champ `value` de `Scalar` est privé, vous ne pouvez pas y accéder de la classe `Set`. Pour cette raison, ajoutez une méthode publique `void set(int value)` à `Scalar` permettant de modifier la valeur du scalaire.

Testez votre opération `Set` en affichant et évaluant l'expression `(set! x 22)` avant d'afficher et évaluer l'expression `(+ x 20)`. Ces affichages et évaluations devraient produire l'affichage suivant :

```
(set! x 22)
22
(+ x 20)
42
```

Aide

L'exécution d'une opération `Set` doit :

- évaluer le membre de droite de l'opération,
- affecter le résultat de cette évaluation à la variable (membre de gauche),
- renvoyer ce résultat.

La méthode `op(0)` renvoie la variable, mais cette dernière a le type `Node`, ce qui fait que vous ne pouvez pas directement invoquer le `set(int value)` de `Variable` pour modifier la valeur de la variable. Pensez alors au transtypage !

Exercice 5 : Vers un interpréteur complet

Vous pouvez maintenant compléter votre petit interpréteur d'arbre pour lui ajouter quelques fioritures comme des structures de contrôle.

Question 5.a De façon à afficher des résultats intermédiaires, ajoutez une classe `Echo` héritant de `Opération`. Cette classe représente une opération (`echo expr`), qui affiche sur le terminal le résultat de l'évaluation de `expr` avant de renvoyer ce résultat. Vérifiez que `(+ (echo (+ 1 2)) 7)` affiche bien 3 pendant l'évaluation et que l'évaluation de l'expression est bien égale à 10.

Question 5.b Pour gérer les conditions, nous avons besoin de booléens. Dans notre petit interpréteur, un scalaire est directement considéré comme un booléen : si sa valeur est égale à 0, c'est que le scalaire est un booléen *faux*, sinon, c'est que c'est un booléen *vrai*. À partir de ce choix, ajoutez une classe `If` permettant d'évaluer une condition. L'arbre d'une condition est `(if condition ifTrue ifFalse)`, où `condition`, `ifTrue` et `ifFalse` sont des noeuds. Vérifiez que `(if 1 42 666)` s'évalue bien en 42 et `(if 0 42 666)` s'évalue bien en 666.

Question 5.c Pour écrire des programmes avancés, nous avons besoin de blocs. Un bloc est simplement une opération dont le nombre d'opérandes est variable et telle que son évaluation évalue chacune des opérandes avant de renvoyer le résultat de l'évaluation de sa dernière opérande. Lorsque nous affichons l'arbre, nous représentons l'expression correspondant à un bloc par `(begin e1 ... en)`. Ajoutez une classe `Begin` permettant d'évaluer un bloc. Vérifiez que `(if 1 (begin (echo 0) 42) 666)` affiche bien 0 pendant l'évaluation et que le résultat de cette évaluation est bien 42.

Question 5.d Ajoutez maintenant la classe `Lt` permettant d'évaluer un arbre (`< e1 e2`). Cette évaluation doit renvoyer vrai (1) si `e1` est plus petit que `e2` et faux (0) sinon.

Question 5.e Finalement, ajoutez une classe `While` permettant d'évaluer l'arbre (`while cond body`). L'évaluation doit commencer par évaluer `cond`. Si le résultat de cette évaluation est vrai (un nombre différent de 0), l'évaluation doit continuer en évaluant `body` avant de recommencer l'évaluation de `cond`. Le résultat de l'évaluation doit être celui du dernier `body` exécuté. Si `body` n'est jamais exécuté, le résultat de l'évaluation doit être le nombre 0.

Construisez, affichez et interprétez l'arbre

```
(begin (set! x 0) (while (< x 10) (begin (echo x) (set! x (+ x 1)))))
```

Qu'affiche votre programme ?

Bonus Pour aller plus loin, vous pouvez ajouter la possibilité de déclarer des fonctions, ou d'autres opérations / structures de contrôle pour améliorer votre interpréteur

Exercice 6 : Tas Binaire et retour au TP 1

Cet exercice n'est à faire que lorsque vous aurez terminé l'ensemble des TP 1, 1_bis et 2. Dans cet exercice, l'objectif est d'implanter une structure de donnée appelée tas binaire.

Un tas binaire est une structure de donnée qui est à la fois :

- Un arbre binaire complet (cf cours de graphe). La complétude signifie que tous les niveaux sauf le dernier sont pleins. Pour le dernier niveau, il est rempli de gauche à droite.
- Un tas, c'est-à-dire que le poids de chaque nœud est inférieur ou égal à celui de chacun de ses fils.

La forme d'arbre, comme souvent, permet d'obtenir des complexités logarithmiques sur les opérations d'accès/modifications. Pour cela, nous allons repartir de la classe `Location` du TP 1, en représentant notre tas sous forme de tableau. Le but à la fin est d'utiliser cette structure de donnée dans notre TP 1 à la place de `LocationSet`.

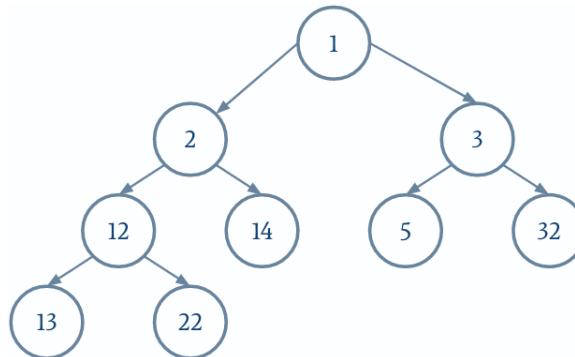


FIGURE 5 – Un exemple de tas binaire.

Question 6.a Créez une nouvelle classe `BinaryHeap` ainsi qu'un constructeur vide pour l'instant.

Question 6.b Copiez le code de la classe `Location` dans votre nouvelle classe. Ajoutez un attribut `positionHeap` qui va représenter la position dans le tas et initialisez-le à -1 . Rajouter un *getter* et un *setter*, c'est-à-dire des méthodes `getPositionHeap` et `setPositionHeap` permettant d'accéder et de modifier la valeur de `positionHeap`.

Question 6.c Le fait qu'un tas soit un arbre binaire complet nous permet donc de le représenter sous forme de tableau. La racine de l'arbre se trouve à la position 0. Si on prend un noeud à un indice i , les fils gauche et droite se trouvent respectivement à l'indice $2 \times i + 1$ et $2 \times i + 2$. Pour cet exercice, nous utilisons donc un tableau extensible de `Location`. Initialiser ce tableau avec une taille initiale de 100 et un indice maximum de 0.

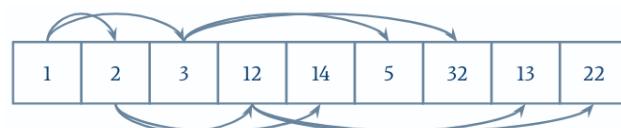


FIGURE 6 – Notre tas binaire représenté sous forme de tableau.

Question 6.d Implémentez une fonction `add` capable de rajouter une `Location` dans le tas. On considérera que la fonction d'ordre entre deux éléments de notre tas est donné par la comparaison de leur distance. Ainsi, dans `add`, on introduit le nouvel élément à la prochaine place libre et on le fait remonter jusqu'à ce qui soit à sa place. On n'oubliera pas de mettre à jour `positionHeap` de chaque `Location`.

Rappel : le parent d'un nœud i est donné par $\frac{i-1}{2}$.

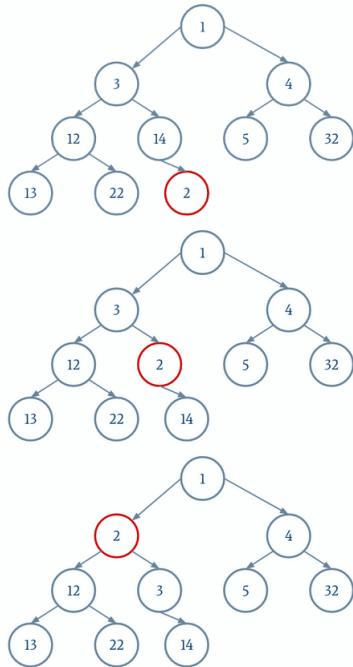


FIGURE 7 – Procédure d'ajout d'une valeur 2 dans notre tas binaire d'exemple.

Question 6.e Quelle est la complexité pire cas de `add` en fonction du nombre de noeuds dans l'arbre ?

Question 6.f L'action de faire remonter une valeur s'appelle une percolation vers le haut. Extrayez une fonction `percolateUp` à partir du code de `add` qui prend en entrée une position et fait remonter le noeud à la bonne position.

Comme la méthode `percolateUp` est une méthode interne, elle ne doit pas être accédée depuis l'extérieur. Faites-en sorte que ce soit le cas.

Vous aurez sûrement envie de créer également une fonction auxiliaire `swap` pour échanger deux indices.

Question 6.g Codez une fonction `removeMin`. Pour cela, on récupère le noeud racine. Cependant, il faut le remplacer par un autre noeud. Pour cela, on prend le dernier noeud du tas, on le met à la place de la racine, et on le fait redescendre jusqu'à la bonne position. La Figure 8 décrit un exemple de cette procédure.

Question 6h Quelle est la complexité pire cas de `removeMin` en fonction du nombre de noeuds dans l'arbre ?

Question 6.i La fonction permettant de faire descendre un noeud jusqu'à sa place s'appelle la percolation vers le bas. Comme pour `add`, extrayez une méthode `percolateDown` de `removeMin`.

Question 6.j Codez une fonction `updateDown` qui met à jour la position d'une `Location` dont on a réduit la distance. Quelle est sa complexité en fonction du nombre de noeuds dans l'arbre ?

Question 6.k Mettez à jour la classe `Location` pour utiliser `BinaryHeap` à la place de `LocationSet`.

Aide

Il faudra bien penser à mettre à jour correctement les `Locations`.

Question 6.1 Quelle est la nouvelle complexité de `findPathTo` ?

Avec notre tas binaire, nous venons en fait de coder une file de priorité. C'est une structure de donnée permettant de récupérer des éléments en fonction d'un poids (la priorité). Le tas binaire est une implémentation possible.

Remarquez que tout ce que nous avons fait ne modifie pas les fonctionnalités de `Location`. Nous avons simplement rendu le code plus rapide. Cela signifie que l'utilisateur de `Location` pourra continuer à utiliser le

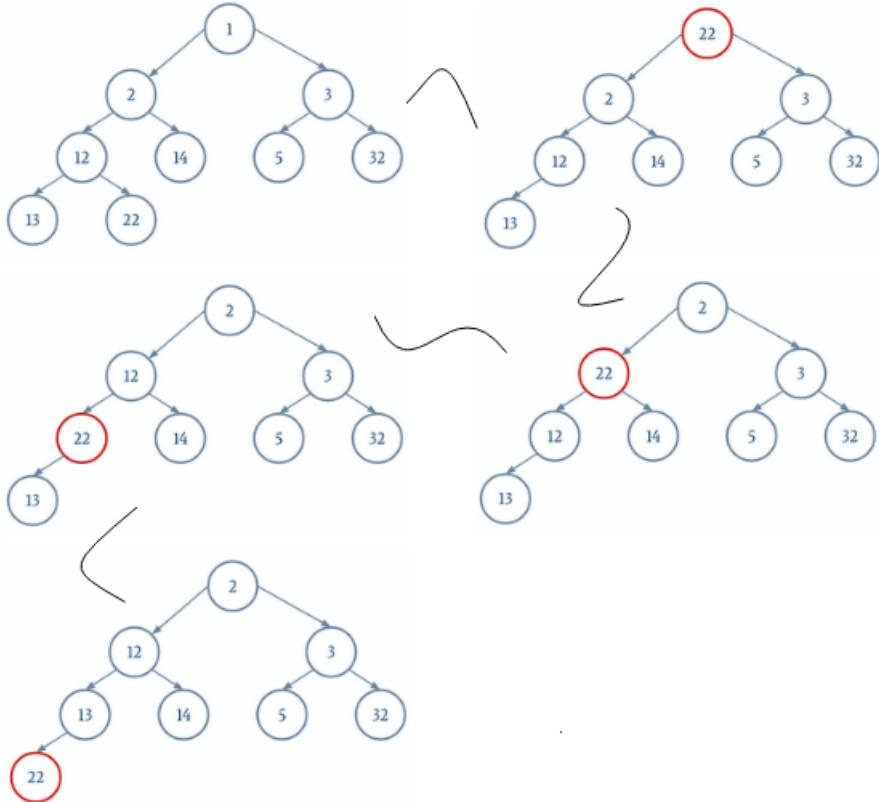


FIGURE 8 – Procédure de suppression de la racine dans notre tas binaire d'exemple.

même code et profitera quand même des améliorations de performance. C'est là la beauté de l'encapsulation ! Nous avons exposé une interface simple (`findPathTo`) et nous avons pu modifier ce qui se passe dans les coulisses, sans conséquences négatives sur les utilisateurs.

Question 6.m Il est encore possible d'améliorer les performances de l'algorithme de Dijkstra en utilisant des Tas de Fibonacci. Codez une version avec ces tas de Fibonacci (cherchez en ligne comment ils fonctionnent).