Chapitre 3

Les arbres

3.1 Introduction

Les arbres sont des structures de données fondamentales et polyvalentes utilisées en informatique et en algorithmique pour représenter et organiser des données de manière hiérarchique. Ils sont largement utilisés pour résoudre une grande variété de problèmes, allant des opérations de recherche et de tri aux problèmes de gestion de données complexes. Dans ce chapitre, nous explorerons en détail les concepts essentiels des arbres, en commençant par les bases de leur structure et de leur terminologie. Nous découvrirons également les différentes variantes d'arbres, notamment les arbres binaires, les arbres de recherche binaires et les arbres équilibrés. Nous étudierons les principales opérations que nous pouvons effectuer sur les arbres, telles que l'insertion, la suppression, la recherche, le parcours et le tri. Nous examinerons également comment optimiser les performances des opérations sur les arbres en utilisant des techniques d'équilibrage pour éviter des temps d'exécution indésirables.

Au fur et à mesure que nous progressons dans ce chapitre, nous aborderons des concepts plus avancés qui permettent d'améliorer davantage les performances des opérations clés sur les arbres. Enfin, nous appliquerons nos connaissances sur les arbres pour résoudre divers problèmes d'algorithmique tels que la recherche efficace, les algorithmes de tri avancés, les parcours d'arbres et d'autres applications intéressantes.

3.2 Rappels d'arbres

Un arbre est une structure de données composée de plusieurs éléments appelés nœuds. Chaque nœud contient une valeur et zéro ou plusieurs nœuds enfants. Un nœud sans enfant est appelé nœud terminal ou feuille. Le nœud à partir duquel tous les autres nœuds sont

accessibles est appelé nœud racine. Les nœuds qui ne sont pas des nœuds racines et qui ont au moins un nœud enfant sont appelés nœuds internes. Un arbre peut être vide, ce qui signifie qu'il ne contient aucun nœud. La figure 3.1 montre un exemple d'arbre. Dans cet arbre, le nœud 1 est la racine, les nœuds 2 et 3 sont des nœuds internes, et les nœuds 4, 5, 6, 7, 8 et 9 sont des nœuds terminaux ou des feuilles.

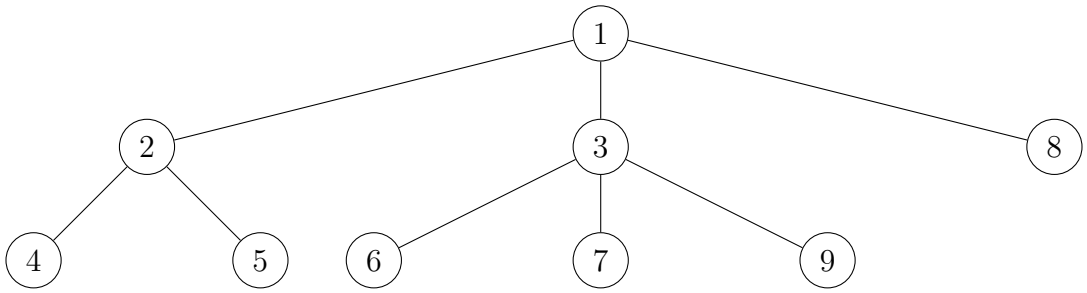

FIGURE 3.1 – Exemple d'arbre

3.3 Arbre binaire

Un arbre binaire est un arbre dans lequel chaque nœud a au plus deux nœuds enfants. Les nœuds enfants sont généralement appelés $filsgauche$ et $filsdroit$. Les arbre binaire ont une importance particulière en informatique car ils peuvent être utilisés pour implémenter efficacement des structures de données telles que les arbres de recherche binaires et les tas binaires. En plus, il est possible de convertir un arbre quelconque en un arbre binaire, ce qui permet d'appliquer des algorithmes spécifiques aux arbres binaires sur n'importe quel arbre.

3.3.1 Parcours d'arbre binaire

Le parcours d'un arbre binaire est une opération qui consiste à visiter chaque nœud de l'arbre exactement une fois. Il existe (principalement) plusieurs façons de parcourir un arbre binaire, nous allons voir les suivants :

— **Parcours préfixe** : Visitez le nœud racine, puis visitez le fils gauche, puis visitez le fils droit.

— **Parcours infixe** : Visitez le fils gauche, puis visitez le nœud racine, puis visitez le fils droit.

— **Parcours postfixe** : Visitez le fils gauche, puis visitez le fils droit, puis visitez le nœud racine.

— **Parcours en largeur** : Visitez les nœuds de chaque niveau de l'arbre de gauche à droite.

Soit l'arbre binaire de la figure 3.2.

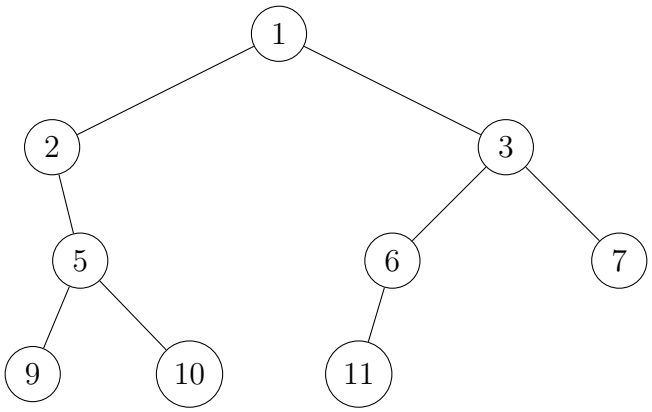

FIGURE 3.2 – Exemple d'arbre binaire

— Le parcours préfixe de cet arbre est : $1, 2, 5, 9, 10, 3, 6, 11, 7$.
— Le parcours infixe est : $2, 9, 5, 10, 1, 11, 6, 3, 7$.
— Le parcours postfixe est : $9, 10, 5, 2, 11, 6, 7, 3, 1$.
— Le parcours en largeur est : $1, 2, 3, 5, 6, 7, 9, 10, 11$.

3.3.2 Arbre binaires particuliers

Les arbres binaires peuvent être classés en fonction de la structure de l'arbre et de la valeur des nœuds. Certains types d'arbres binaires sont plus adaptés à certains problèmes que d'autres. Dans cette section, nous allons voir les types d'arbres binaires les plus courants.

Arbre binaire strict

Un arbre binaire strict est un arbre binaire dans lequel chaque nœud a exactement deux nœuds enfants ou aucun nœud enfant. Aucun nœud ne peut avoir un seul nœud enfant. Un arbre binaire strict est également appelé arbre binaire localement complet. Un arbre binaire strict possède $2P + 1$ nœuds, dont P nœuds internes et $P + 1$ nœuds feuilles. La figure 3.3 montre un exemple d'arbre binaire strict.

Arbre binaire parfait

Un arbre binaire parfait est un arbre binaire strict dans lequel tous les nœuds internes ont exactement deux nœuds enfants et toutes les feuilles sont situées au même niveau (ont

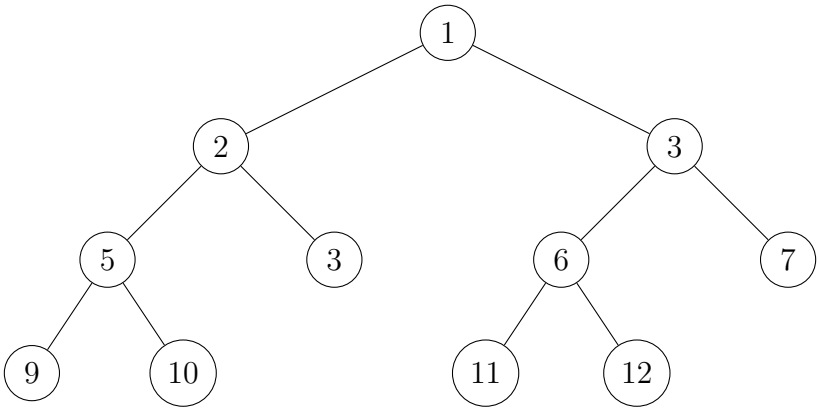

FIGURE 3.3 – Exemple d'arbre binaire strict

la même hauteur). Un arbre binaire parfait de hauteur H possède exactement $2^{H+1} - 1$ nœuds, dont $2^H - 1$ nœuds internes et 2^H feuilles. La figure 3.4 montre un exemple d'arbre binaire parfait.

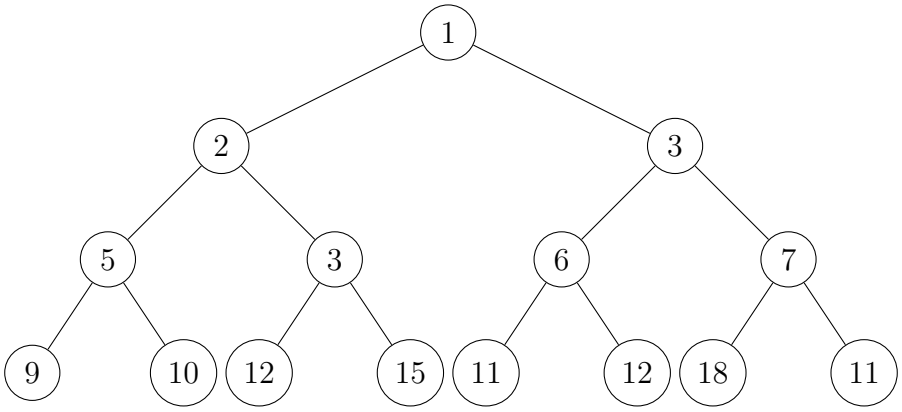

FIGURE 3.4 – Exemple d'arbre binaire parfait

Arbre binaire équilibré

Un arbre binaire est dit équilibré si la différence de hauteur entre ses fils gauche et droit à chaque nœud est au plus 1. Cela garantit que la hauteur de l'arbre est de $O(\log n)$, où n est le nombre de nœuds de l'arbre, ce qui assure des opérations efficaces sur l'arbre et des performances équilibrées même en cas de parallelisation.

Arbre binaire complet

Un arbre binaire complet est un arbre binaire strict dans lequel tous les nœuds internes ont exactement deux nœuds enfants et toutes les feuilles sont situées au même niveau

(ont la même hauteur), sauf peut-être les feuilles les plus à droite. Autrement dit, un arbre binaire complet est un arbre binaire parfait dans lequel les feuilles les plus à droite peuvent être manquantes, ou un arbre binaire à la fois strict et équilibré. La figure 3.5 montre un exemple d'arbre binaire complet.

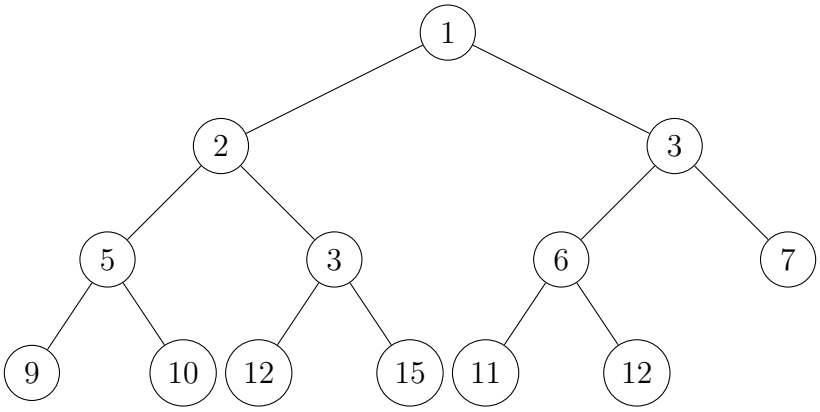

FIGURE 3.5 – Exemple d'arbre binaire complet

Arbre binaire de recherche

Un arbre binaire de recherche est un arbre binaire dans lequel chaque nœud a une valeur supérieure ou égale à toutes les valeurs qui se trouvent sur son fils gauche et une valeur inférieure ou égale à toutes les valeurs qui se trouvent sur son fils droit. Un parcours infixe d'un arbre binaire de recherche donne forcément une liste de valeurs triées par ordre croissant. La figure 3.6 montre un exemple d'arbre binaire de recherche.

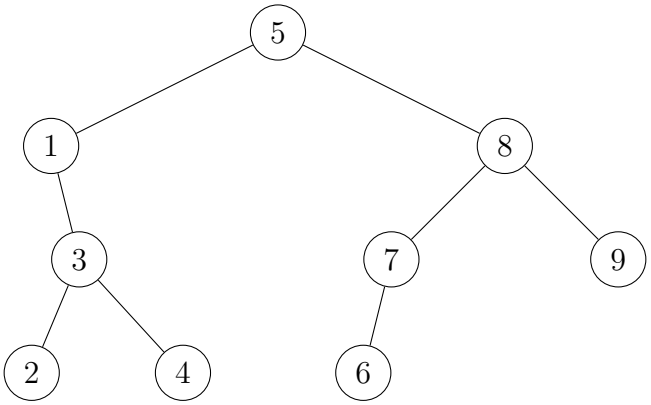

FIGURE 3.6 – Exemple d'arbre binaire de recherche

Les tas

Un tas est un arbre binaire complet dans lequel chaque nœud a une valeur supérieure/inférieure ou égale à toutes les valeurs qui se trouvent sur ses fils. Un tas est également appelé arbre binaire de tas. On peut distinguer deux types de tas :

— **Le tas max** : Dans un tas max, la valeur de chaque nœud est supérieure ou égale à toutes les valeurs qui se trouvent sur ses fils.

— **Le tas min** : Dans un tas min, la valeur de chaque nœud est inférieure ou égale à toutes les valeurs qui se trouvent sur ses fils.

La figure 3.7 montre un exemple de tas min.

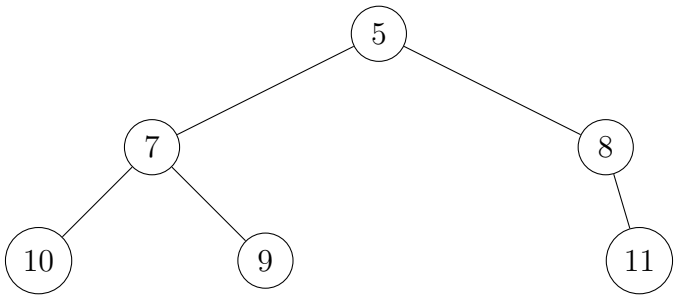

FIGURE 3.7 – Exemple de tas min

3.3.3 Représentation d'un arbre quelquonque sous forme d'un arbre binaire

Un arbre quelconque A peut être représenté sous forme d'un arbre binaire B tel que :

— Chaque nœud de B représente un nœud de A.

— Le fils gauche d'un nœud de B représente le premier fils du nœud correspondant de A.

— Le fils droit d'un nœud de B représente le frère droit du nœud correspondant de A.

L'arbre dans la figure 3.1 qui n'est pas binaire, peut être représenté sous forme d'un arbre binaire comme il est montré dans la figure 3.8.

L'approche utilisée pour cette conversion est apppelée **arbre fils frère**. L'utilité de cette conversion réside dans le fait que les opérations sur les arbres binaires telles que la recherche, l'insertion et la suppression sont plus faciles à implémenter que sur les arbres quelconques. En convertissant un arbre général en un arbre binaire, on peut appliquer ces opérations plus facilement et profiter des avantages des arbres binaires pour résoudre des

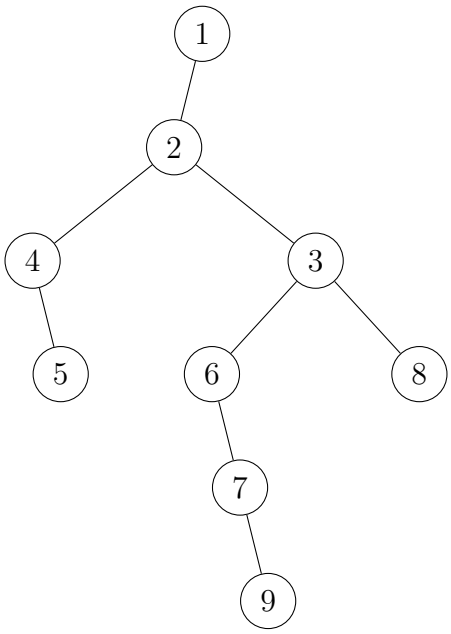

FIGURE 3.8 – Conversion de l'arbre de la figure 3.1 en un arbre binaire

problèmes spécifiques. Cela justifie le choix de se concentrer principalement sur les arbres binaires dans la suite de ce chapitre.

3.4 Implémentations

Nous allons montrer dans cette section comment déclarer la structure, et implémenter les opérations de base sur un arbre. Nous allons d'abord commencer par les arbres généraux, puis nous allons nous concentrer sur les différents types d'arbres binaires.

3.4.1 Arbre général : primitives et implémentation

Pour implémenter un arbre général, on a besoin de définir une structure qui représente un nœud de l'arbre. Cette structure doit contenir les informations suivantes :

— **La valeur du nœud** : La valeur du nœud peut être de n'importe quel type. Dans notre cas, nous allons supposer que la valeur est de type **int**, mais elle peut être de n'importe quel autre type simple ou composé.

— **Les fils du nœud** : Les nœuds d'un arbre général peuvent avoir un nombre quelconque de fils. Pour représenter les fils d'un nœud, n'importe quelle collection de données de taille dynamique peut être utilisée. Vu que nous nous concentrons principalement sur l'implémentation en langage C, nous allons utiliser une liste chaînée

pour représenter les fils d'un nœud. Cette dernière peut contenir un nombre quelconque de nœuds, et chaque nœud de la liste représente un fils du nœud courant.

La structure d'un arbre général est définie comme suit :

```
1  typedef struct Node *Tree;
2  typedef struct ListTreeNode *ListTree;
3  typedef struct ListTreeNode {
4    Tree current;
5    ListTree next;
6  } ListTreeNode;
7  typedef struct Node {
8    int value;
9    ListTree children;
10 } Node;
```

Algorithm 3.1 – Structure d'un nœud

L'arbre général est défini par le type **Tree** qui est un pointeur vers un nœud, dans le cas où l'arbre est vide, le pointeur est nul. Un nœud est défini par le type **Node** qui contient la valeur du nœud et une liste de ses fils, qui est définie par le type **ListTree**, une liste chainée contenant plusieurs arbres généraux. Il est évidement possible et même plus simple d'utiliser une structure basée sur les arbres binaires pour implémenter un arbre général, mais nous avons choisi cette approche pour montrer les différentes possibilités d'implémentation. Les arbres binaires serons vus plus en détail dans la section 3.4.2.

Nous montrons ci-dessous comment implémenter quelques opérations de base sur un arbre général. La structure **Tree** est définie dans l'algorithme 3.1, mais vous pouvez adapter les implémentations pour une autre structure si vous le souhaitez.

Recherche d'une valeur dans un arbre général

Pour chercher une valuer dans un arbre, il faut d'abord penser à une stratégie de parcours. Dans le cas d'un arbre général, il existe plusieurs stratégies de parcours, y compris le parcours en largeur et le parcours en profondeur. Nous allons implémenter la recherche en profondeur, qui est plus simple à implémenter que le parcours en largeur qui nécessite l'utilisation d'une file pour stocker les nœuds à visiter. La recherche en profondeur peut être implémentée récursivement ou itérativement. Nous allons implémenter la version récursive, mais il est possible d'implémenter la version itérative en utilisant une pile pour stocker les nœuds à visiter. La version récursive consiste à visiter le nœud courant, puis visiter récursivement chacun de ses fils. La recherche s'arrête lorsque la valeur est trouvée ou lorsque tous les nœuds de l'arbre sont visités. L'implémentation de la recherche récursive est donnée dans le code suivant :

```
1  int searchTree(Tree tree, int value) {
2    if (tree == NULL) return 0;
3    if (tree->value == value) return 1;
4    ListTree children = tree->children;
5    while (children != NULL) {
6      if (searchTree(children->current, value)) return 1;
7      children = children->next;
8    }
9    return 0;
10 }
```

Algorithm 3.2 – Recherche d'une valeur dans un arbre général - Parcours en profondeur

Une implémentation itérative du parcours en profondeur ainsi qu'une implémentation du parcours en largeur serons données pour les arbres binaires dans la section 3.4.2, mais l'idée peut simplement être généralisée pour les arbres généraux.

Vider un arbre général

Pour vider un arbre général, il suffit de libérer la mémoire allouée pour chaque nœud de l'arbre. Pour chaque nœud, il faut d'abord libérer la mémoire allouée pour ses fils, puis libérer la mémoire allouée pour le nœud lui-même. L'implémentation de la fonction **freeTree** est donnée dans le code ci-dessous :

```
1  void freeTree(Tree tree) {
2    if (tree != NULL) {
3      ListTree children = tree->children;
4      while (children != NULL) {
5        freeTree(children->current);
6        ListTree tmp = children;
7        children = children->next;
8        free(tmp);
9      }
10     free(tree);
11   }
12 }
```

Algorithm 3.3 – Vider un arbre général

Calculer la hauteur d'un arbre général

La hauteur d'un arbre est donné par la longueur du plus long chemin entre la racine et une feuille. Pour calculer la hauteur d'un arbre général, il faut calculer la hauteur de chacun de ses fils, puis prendre le maximum de ces hauteurs et ajouter 1. Nous implémentons cette fonction récursivement comme suit :

```
1  int heightTree(Tree tree) {
2    if (tree == NULL) return 0;
3    int max = 0;
4    ListTree children = tree->children;
5    while (children != NULL) {
6      int height = heightTree(children->current);
7      if (height > max) max = height;
8      children = children->next;
9    }
10   return max + 1;
11 }
```

Algorithm 3.4 – Calculer la hauteur d'un arbre général

Plusieurs autres opérations peuvent être implémentées sur les arbres généraux, comme la recherche du nombre de nœuds, la recherche du nombre de feuilles, la recherche du nombre de nœuds internes, etc. Nous ne détaillons pas ces opérations dans ce chapitre, mais ça sera un bon exercice de les implémenter vous même.

3.4.2 Arbres binaires

Un arbre binaire est un arbre dans lequel chaque nœud possède au plus deux fils. Un arbre binaire peut être vide, ou il peux contenir un nœud racine, ce dernier peut avoir un fils gauche et un fils droit. Chacun de ces fils est aussi un arbre binaire. La structure d'un arbre binaire est plus simple, elle est donnée ci-dessous.

```
1  typedef struct Node *Tree;
2  typedef struct Node {
3    int value;
4    Tree left;
5    Tree right;
6  } Node;
```

Algorithm 3.5 – Structure d'un arbre binaire

Un arbre binaire est défini par le type **Tree** qui est un pointeur vers un nœud, dans le cas où l'arbre est vide, le pointeur est nul. Un nœud est défini par le type **Node** qui contient la valeur du nœud et deux pointeurs vers ses fils gauche et droit. Nous passons maintenant à l'implémentation de quelques opérations de base sur les arbres binaires. Pour éviter de répéter les idées, nous choisissons des fonctions différentes de celles vues pour les arbres généraux, ou au moins nous changeons la façon dont elles sont implémentées.

Recherche d'une valeur dans un arbre binaire - Versions itératives

Nous allons voir ici comment chercher une valeur dans un arbre binaire, cette idée a déjà été explorée dans la section 3.4.1, l'objectif ici est de se passer de la version récursive (qui est plus simple à implémenter) et d'implémenter la version itérative pour varier la façon d'approcher le problème. Nous allons voir deux versions de la recherche itérative, la première est basée sur le parcours en profondeur, et la deuxième est basée sur le parcours en largeur.

Parcours en profondeur Pour implémenter la recherche en profondeur, nous utilisons une pile pour stocker les nœuds à visiter. Nous commençons par empiler la racine de l'arbre, puis nous entrons dans une boucle qui se répète tant que la pile n'est pas vide. Dans cette boucle, nous allons dépiler un nœud, le visiter, puis empiler ses fils droit et gauche s'il en a (tous ses fils dans le cas d'un arbre général). Si le nœud dépilé est celui que nous cherchons, nous retournons vrai, sinon nous continuons la boucle. Si la boucle se termine sans que nous trouvions le nœud, nous retournons faux. L'implémentation de cette fonction suppose l'existance d'un type **Stack** qui est une pile d'arbres (pointeurs vers des nœuds), et des fonctions **init**, **push**, **pop** et **isEmpty** qui permettent respectivement d'initialiser une pile vide, d'empiler un élément, de dépiler un élément et de tester si la pile est vide. Le code de la fonction **searchTree** est donné ci-dessous :

```
1  int searchTree(Tree tree, int value) {
2    if (tree == NULL) return 0;
3    Stack stack = init();
4    push(stack, tree);
5    while (!isEmpty(stack)) {
6      Tree current = pop(stack);
7      if (current->value == value) return 1;
8      if (current->right != NULL) push(stack, current->right);
9      if (current->left != NULL) push(stack, current->left);
10   }
11   return 0;
12 }
```

Algorithm 3.6 – Recherche d'une valeur dans un arbre binaire - Parcours en profondeur (Version itérative)

Parcours en largeur Le parcours en largeur utilise une file d'attente au lieu d'une pile. Cela permet de visiter les nœuds par niveau, c'est-à-dire que nous commençons par visiter la racine, puis nous visitons tous les nœuds du premier niveau (car serons les premiers à être ajoutés à la file), puis nous visitons tous les nœuds du deuxième niveau, etc. L'implémentation de cette fonction suppose l'existance d'un type **Queue** qui est

une file d'arbres, et des fonctions **init**, **enqueue**, **dequeue** et **isEmpty** qui permettent respectivement d'initialiser une file vide, d'ajouter un élément à la fin de la file, de retirer un élément du début de la file et de tester si la file est vide.

```
1  int searchTree(Tree tree, int value) {
2    if (tree == NULL) return 0;
3    Queue queue = init();
4    enqueue(queue, tree);
5    while (!isEmpty(queue)) {
6      Tree current = dequeue(queue);
7      if (current->value == value) return 1;
8      if (current->left != NULL) enqueue(queue, current->left);
9      if (current->right != NULL) enqueue(queue, current->right);
10   }
11   return 0;
12 }
```

Algorithm 3.7 – Recherche d'une valeur dans un arbre binaire - Parcours en largeur

Vérification si un arbre binaire est équilibré

Un arbre binaire est équilibré si la différence entre la hauteur de son fils gauche et celle de son fils droit est au plus 1 à chaque nœud. Pour vérifier si un arbre binaire est équilibré, nous allons utiliser une fonction récursive qui calcule la hauteur d'un arbre, et nous allons vérifier la condition d'équilibre à chaque nœud. La fonction **isBalanced** est donnée ci-dessous :

```
1  int height(Tree tree) {
2    if (tree == NULL) return 0;
3    int leftHeight = height(tree->left);
4    int rightHeight = height(tree->right);
5    return 1 + (leftHeight > rightHeight ? leftHeight : rightHeight);
6  }
7
8  int isBalanced(Tree tree) {
9    if (tree == NULL) return 1;
10   int leftHeight = height(tree->left);
11   int rightHeight = height(tree->right);
12   if (abs(leftHeight - rightHeight) > 1) return 0;
13   return isBalanced(tree->left) && isBalanced(tree->right);
14 }
```

Algorithm 3.8 – Vérification si un arbre binaire est équilibré

Dans l'implémentation ci-dessus, nous calculons la hauteur de l'arbre à chaque nœud, ce qui est inefficace. Nous pouvons améliorer l'efficacité de l'algorithme en calculant la

hauteur de l'arbre et en vérifiant la condition d'équilibre en même temps. Pour cela, nous allons utiliser une valeur de retour spéciale pour indiquer que l'arbre n'est pas équilibré. La valeur de retour de la fonction **isBalanced** est un entier qui représente la hauteur de l'arbre, et si l'arbre n'est pas équilibré, la fonction retourne -1. Le code de la fonction **isBalanced** est donné ci-dessous :

```
1  int isBalanced(Tree tree) {
2    if (tree == NULL) return 0;
3    int leftHeight = isBalanced(tree->left);
4    if (leftHeight == -1) return -1;
5    int rightHeight = isBalanced(tree->right);
6    if (rightHeight == -1) return -1;
7    if (abs(leftHeight - rightHeight) > 1) return -1;
8    return 1 + (leftHeight > rightHeight ? leftHeight : rightHeight);
9  }
```

Algorithm 3.9 – Vérification si un arbre binaire est équilibré (Version améliorée)

3.4.3 Arbre binaire de recherche

Un arbre binaire de recherche est un arbre binaire dans lequel chaque nœud a une valeur qui est supérieure ou égale à toutes les valeurs sur son fils gauche et inférieure ou égale à toutes les valeurs sur son fils droit comme nous avons montré dans la figure 3.6. La structure des données dans un arbre binaire de recherche n'est pas différente de celle d'un arbre binaire. On doit seulement s'assurer que la distribution des valeurs dans l'arbre respecte la condition d'un arbre binaire de recherche. Pour cela, nous allons utiliser la même structure dans l'algorithme 3.5 pour représenter un arbre binaire, puis implémenter une fonction **isBinarySearchTree** qui permet de vérifier si un arbre binaire donné est un arbre binaire de recherche ou pas.

Vérification si un arbre binaire est un arbre binaire de recherche

Pour vérifier qu'un arbre binaire est un arbre binaire de recherche, nous allons d'abord vérifier que son fils gauche et fils droit sont des arbres binaires de recherche, puis nous allons vérifier que la valeur du nœud est supérieure ou égale à toutes les valeurs sur son fils gauche et inférieure ou égale à toutes les valeurs sur son fils droit. Noter qu'une fois que nous avons vérifié qu'un arbre binaire est un arbre binaire de recherche, la recherche de son minimum et/ou de son maximum est triviale, il suffit de suivre le chemin le plus à gauche pour trouver le minimum et le chemin le plus à droite pour trouver le maximum. L'implémentation de la fonction **isBinarySearchTree** est donnée ci-dessous :

```
1  int maxSearchTree(Tree tree) {
2    if (tree->right == NULL) return tree->value;
```

```
3     return maxSearchTree(tree->right);
4   }
5
6   int minSearchTree(Tree tree) {
7     if (tree->left == NULL) return tree->value;
8     return minSearchTree(tree->left);
9   }
10
11  int isBinarySearchTree(Tree tree) {
12    if (tree == NULL) return 1;
13    if (!isBinarySearchTree(tree->left)
14        || !isBinarySearchTree(tree->right)) return 0;
15    if (tree->left != NULL && maxSearchTree(tree->left) > tree->value)
16        return 0;
17    if (tree->right != NULL && minSearchTree(tree->right) < tree->value)
18        return 0;
19    return 1;
20  }
```

Algorithm 3.10 – Vérification si un arbre binaire est un arbre binaire de recherche

Comme nous avons fait pour la fonction **isBalanced**, nous pouvons améliorer l'efficacité de la fonction **isBinarySearchTree** en utilisant deux autres paramètres pour garder une trace de la valeur maximale et minimale de l'arbre à chaque étape. Cela évite de parcourir les fils gauche et droit de chaque nœud une deuxième fois pour trouver leur maximum et minimum. Le code de la fonction **isBinarySearchTree** est donné ci-dessous :

```
1   int isBinarySearchTree(Tree tree, int min, int max) {
2     if (tree == NULL) return 1;
3     if (tree->value < min || tree->value > max) return 0;
4     return isBinarySearchTree(tree->left, min, tree->value)
5         && isBinarySearchTree(tree->right, tree->value, max);
6   }
```

Algorithm 3.11 – Vérification si un arbre binaire est un arbre binaire de recherche (Version améliorée)

Cette fonction doit être appelée initialement avec les valeurs INT_MIN et INT_MAX comme paramètres pour le minimum et le maximum. Par la suite, le minimum et le maximum sont mis à jour à chaque étape. Losrque nous appelons la fonction **isBinarySearchTree** sur le fils gauche, le maximum est mis à jour avec la valeur du nœud courant, en d'autres termes, toutes les valeurs sur le fils gauche doivent être inférieures ou égales à la valeur du nœud courant. Et lorsque nous appelons la fonction **isBinarySearchTree** sur le fils droit, le minimum est mis à jour avec la valeur du nœud courant, en d'autres termes, toutes les valeurs sur le fils droit doivent être supérieures ou égales à la valeur du

nœud courant. Cette condition est testée récurivement dans la ligne 3 du listing 3.11. Ou idéalement, cette fonction doit être utilisée comme utilitaire pour une fonction principale qui ne prend qu'un seul paramètre, l'arbre binaire.

Insertion et recherche dans un arbre binaire de recherche

L'insertion dans un arbre binaire de recherche doit respecter les conditions que nous avons déja énoncées. En d'autres termes, la valeur à insérer doit parcourir le chemin le plus à gauche si elle est inférieure à la valeur du nœud courant, et le chemin le plus à droite si elle est supérieure. L'inserion se fait dés lors que nous atteignons un nœud dont le fils gauche ou droit est NULL. Le code de la fonction **insert** est donné ci-dessous :

```
1  Tree createNode(int value) {
2    Tree tree = (Tree) malloc(sizeof(struct Node));
3    tree->value = value;
4    tree->left = NULL;
5    tree->right = NULL;
6    return tree;
7  }
8
9  Tree insert(Tree tree, int value) {
10   if (tree == NULL) {
11     tree = createNode(value);
12   } else if (value < tree->value) {
13     tree->left = insert(tree->left, value);
14   } else {
15     tree->right = insert(tree->right, value);
16   }
17   return tree;
18 }
```

Algorithm 3.12 – Insertion dans un arbre binaire de recherche

La fonction **createNode** est une fonction utilitaire qui permet de créer un nœud avec une valeur donnée et un fils gauche et droit NULL. La fonction **insert** est une fonction récursive qui prend en paramètre un arbre binaire de recherche et une valeur à insérer. Si l'arbre binaire de recherche est NULL, alors nous créons un nœud avec la valeur donnée. Sinon, nous comparons la valeur à insérer avec la valeur du nœud courant pour décider quelle branche prendre.

Le principe de la recherche dans un arbre binaire de recherche est le même que celui de l'insertion. Nous parcourons le chemin le plus à gauche si la valeur à rechercher est inférieure à la valeur du nœud courant, et le chemin le plus à droite si elle est supérieure. La recherche s'arrête si nous atteignons un nœud dont la valeur est égale à la valeur

recherchée ou si nous atteignons un nœud dont le fils gauche ou droit est NULL. Nous donnons ci-dessous le code de la fonction **search** :

```
1  int search(Tree tree, int value) {
2    if (tree == NULL) return 0;
3    if (tree->value == value) return 1;
4    if (value < tree->value) return search(tree->left, value);
5    return search(tree->right, value);
6  }
```

Algorithm 3.13 – Recherche dans un arbre binaire de recherche

Suppression dans un arbre binaire de recherche

La suppression dans un arbre binaire de recherche est un peu plus complexe que l'insertion et la recherche. Nous devons d'abord trouver le nœud à supprimer, puis distinguer plusieurs cas possibles.

- Le premier cas est lorsque le nœud à supprimer est une feuille, c'est-à-dire qu'il n'a pas de fils gauche ou droit. Dans ce cas, nous pouvons simplement supprimer le nœud.
- Le deuxième cas est lorsque le nœud à supprimer a un seul fils, soit le fils gauche, soit le fils droit, mais pas les deux. Dans ce cas, nous pouvons simplement supprimer le nœud et le remplacer par son fils comme nous ferions pour une liste chaînée.
- Le cas le plus complexe est lorsque le nœud à supprimer a deux fils. Dans ce cas, nous devons d'abord trouver son successeur, c'est-à-dire le minimum de son fils droit. Ensuite, nous remplaçons la valeur du nœud à supprimer par la valeur de son successeur, et finalement nous supprimons le successeur du fils droit. Nous aurions pu également trouver le prédécesseur du nœud à supprimer, c'est-à-dire le maximum de son fils gauche, et faire la même chose.

Le code de la fonction **delete** est donné ci-dessous :

```
1  Tree delete(Tree tree, int value) {
2    if (tree == NULL) return NULL;
3    if (value < tree->value) {
4      tree->left = delete(tree->left, value);
5    } else if (value > tree->value) {
6      tree->right = delete(tree->right, value);
7    } else {
8      if (tree->left == NULL) {
9        Tree temp = tree->right;
10       free(tree);
```

```
11          return temp;
12        } else if (tree->right == NULL) {
13          Tree temp = tree->left;
14          free(tree);
15          return temp;
16        }
17        int successor = minSearchTree(tree->right);
18        tree->value = successor;
19        tree->right = delete(tree->right, successor);
20      }
21      return tree;
22    }
```

Algorithm 3.14 – Suppression dans un arbre binaire de recherche

La fonction **minSearchTree** est implémentée dans précédemment dans l'algorithme 3.10. La fonction **delete** est une fonction récursive qui prend en paramètre un arbre binaire de recherche et une valeur à supprimer. Si l'arbre binaire de recherche est NULL, alors nous retournons NULL. Sinon, nous comparons la valeur à supprimer avec la valeur du nœud courant pour décider quelle branche prendre. Si la valeur du nœud est égal à la valeur à supprimer, alors nous distinguons les trois cas que nous avons énoncés précédemment.

3.4.4 Arbre binaire de recherche équilibré

Comme son nom l'indique, c'est un arbre binaire de recherche dont la hauteur est équilibrée. Cela signifie que la différence de hauteur entre le sous-arbre gauche et le sous-arbre droit est au plus 1 pour chaque nœud. Toutes les opérations d'inserion, et de suppression sur un arbre binaire de recherche équilibré doivent tenir compte et maintenir cette propriété. Très souvent on utilise une opération de rotation si nécessaire pour maintenir cette propriété. La structure des données dans un arbre binaires de recherche équilibré est la même que ce qu'on a vu précédemment pour un arbre binaire de recherche, on doit seulement maintenir la propriété d'équilibre dans toutes les opérations d'insertion et de suppression.

Rotation

La rotation est une opération qui permet de maintenir la propriété d'équilibre dans un arbre binaire de recherche équilibré. Il existe deux types de rotation :

— La rotation simple à gauche : elle permet de rééquilibrer un arbre binaire de recherche équilibré lorsque la différence de hauteur entre le sous-arbre gauche et le sous-arbre droit est de -2.

— La rotation simple à droite : elle permet de rééquilibrer un arbre binaire de recherche équilibré lorsque la différence de hauteur entre le sous-arbre gauche et le sous-arbre droit est de 2.

La figure 3.9 illustre la rotation simple à droite. Dans cet exemple, la différence de hauteur entre le sous-arbre gauche et le sous-arbre droit est de 2. La rotation simple à droite consiste à changer la racine de l'arbre par son fils gauche, et à déplacer le fils droit du fils gauche de la racine vers le fils gauche de l'ancienne racine.

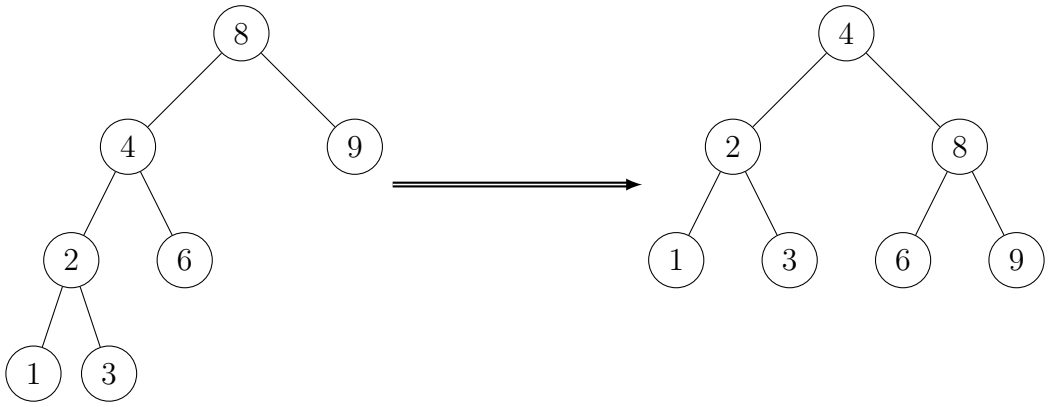

FIGURE 3.9 – Rotation simple à droite

Notez qu'après la rotation, l'arbre reste un arbre binaire de recherche, mais le fils gauche et le fils droit de la racine ont désormais la même hauteur. La rotation simple à gauche est similaire à la rotation simple à droite, mais dans le sens inverse. Il est parfois nécessaire de faire une rotation simple à gauche et/ou à droite plusieurs fois et sur des niveaux différents pour rééquilibrer un arbre binaire de recherche. La rotation doit être effectuée si nécessaire après chaque insertion et suppression dans un arbre binaire de recherche équilibré pour qu'il garde sa propriété d'équilibre.

L'implémentation de la rotation simple à droite et à gauche dons données dans le code ci-dessous :

```
1  Tree rightRotation(Tree tree) {
2    Tree newRoot = tree->left;
3    tree->left = newRoot->right;
4    newRoot->right = tree;
5    return newRoot;
6  }
7
8  Tree leftRotation(Tree tree) {
9    Tree newRoot = tree->right;
10   tree->right = newRoot->left;
```

```
11    newRoot->left = tree;
12    return newRoot;
13  }
```

Algorithm 3.15 – Rotation simple à droite

Insertion dans un arbre binaire de recherche équilibré

L'insertion dans un arbre binaire de recherche équilibré est similaire à l'insertion dans un arbre binaire de recherche. La seule différence est qu'après l'insertion, il faut vérifier si la propriété d'équilibre est toujours respectée. Si ce n'est pas le cas, il faut faire une rotation simple à droite ou à gauche pour rééquilibrer l'arbre.

```
1   Tree balancedInsert(Tree tree, int value) {
2     if (tree == NULL) return createNode(value);
3     if (value < tree->value) {
4       tree->left = balancedInsert(tree->left, value);
5       if (height(tree->left) - height(tree->right) == 2) {
6         tree = rightRotation(tree);
7       }
8       return tree;
9     }
10    else {
11      tree->right = balancedInsert(tree->right, value);
12      if (height(tree->right) - height(tree->left) == 2) {
13        tree = leftRotation(tree);
14      }
15      return tree;
16    }
17  }
```

Algorithm 3.16 – Insertion dans un arbre binaire de recherche équilibré

La fonction `balancedInsert` prend en paramètre un arbre binaire de recherche équilibré et une valeur à insérer dans l'arbre. L'insertion se fait comme dans un arbre binaire de recherche, mais après l'insertion, on vérifie si la propriété d'équilibre est toujours respectée. Vu que l'arbre est équilibré avant l'insertion, la différence de hauteur entre le sous-arbre gauche et le sous-arbre droit est de 0 ou 1. Si la différence de hauteur entre le sous-arbre gauche et le sous-arbre droit est de 2, cela veut dire que l'arbre n'est plus équilibré. Dans ce cas, on fait une rotation simple à droite ou à gauche pour rééquilibrer l'arbre. La rotation est effectuée sur le nœud où la différence de hauteur entre le sous-arbre gauche et le sous-arbre droit est de 2.

Suppression d'un arbre binaire de recherche équilibré

La suppression dans un arbre binaire de recherche équilibré est similaire à la suppression dans un arbre binaire de recherche. Mais comme pour l'insertion, il faut vérifier si la propriété d'équilibre est toujours respectée après la suppression. Si ce n'est pas le cas, il faut faire une rotation simple à droite ou à gauche pour rééquilibrer l'arbre. La fonction `balancedDelete` ci-dessous illustre la suppression dans un arbre binaire de recherche équilibré.

```
1  Tree balancedDelete(Tree tree, int value) {
2    if (tree == NULL) return NULL;
3    if (value < tree->value) {
4      tree->left = balancedDelete(tree->left, value);
5      if (height(tree->right) - height(tree->left) == 2)
6        tree = leftRotation(tree);
7      return tree;
8    }
9    else if (value > tree->value) {
10     tree->right = balancedDelete(tree->right, value);
11     if (height(tree->left) - height(tree->right) == 2)
12       tree = rightRotation(tree);
13     return tree;
14   }
15   else {
16     if (tree->left == NULL && tree->right == NULL) {
17       free(tree);
18       return NULL;
19     }
20     else if (tree->left == NULL) {
21       Tree temp = tree->right;
22       free(tree);
23       return temp;
24     }
25     else if (tree->right == NULL) {
26       Tree temp = tree->left;
27       free(tree);
28       return temp;
29     }
30     else {
31       int successor = minSearchTree(tree->right);
32       tree->value = successor;
33       tree->right = balancedDelete(tree->right, successor);
34       if (height(tree->left) - height(tree->right) == 2) {
35         tree = rightRotation(tree);
36       }
37       return tree;
```

```
38        }
39      }
40    }
```

Algorithm 3.17 – Suppression dans un arbre binaire de recherche équilibré

3.5 Structure de données Tas

3.5.1 Définition

Un tas est un arbre binaire complet où chaque nœud est plus grand que ses fils. Un arbre binaire complet est un arbre binaire où tous les niveaux sont remplis, sauf peut-être le dernier niveau. Si le dernier niveau n'est pas rempli, les nœuds sont remplis de gauche à droite.

3.5.2 Implémentation d'un arbre binaire complet

Un arbre binaire complet peut être implémenté avec un tableau. Pour un nœud à l'indice i dans un tableau indexé à partir de 0, son fils gauche est à l'indice $2i+1$ et son fils droit est à l'indice $2i+2$. Le nœud à l'indice i a pour parent le nœud à l'indice $\lfloor \frac{i-1}{2} \rfloor$. La déclaration d'un arbre binaire complet avec un tableau est donnée dans le code ci-dessous :

```
1  typedef struct CompleteBinaryTree {
2    int *array;
3    int size;
4  } CompleteBinaryTree;
```

Algorithm 3.18 – Déclaration d'un arbre binaire complet avec un tableau

La structure `CompleteBinaryTree` contient un tableau d'entiers et la taille du tableau. La taille du tableau est le nombre de nœuds effectivement présents dans le tableau, ce dernier peut être plus grand que la valeur du champs `size`, ce qui permet d'ajouter plus de nœuds à l'arbre binaire complet sans avoir à redimensionner le tableau. Initialement la taille du tableau est 0.

3.5.3 Tri par tas

Le tri par tas est un algorithme de tri basé sur la structure de données du tas binaire. Dans un tas binaire min, chaque nœud a une valeur inférieure ou égale à celle de ses nœuds enfants. Les étapes de l'algorithme de tri par tas sont les suivantes :

1. Construire un tas binaire min à partir du tableau à trier, pour ce faire, nous commençons par construire un tas binaire min contenant un seul élément, puis nous

insérons les éléments du tableau à trier un par un dans le tas. L'insertion d'un élément dans un tas binaire est expliquée dans la section 3.5.4.

2. Extraire le minimum du tas binaire (la racine de l'arbre, ou le premier élément du tableau) et l'insérer à la fin d'un tableau qui contiendra les éléments triés. Après l'extraction du minimum, le tas doit être réorganisé. Nous utilisons un processus appelé *percolation vers le bas* qui est expliqué dans la section 3.5.5.
3. Répéter l'étapes 2 jusqu'à ce que le tas binaire soit vide.

3.5.4 Insertion dans un tas binaire

L'insertion dans un tas binaire se fait en insérant l'élément à la fin du tableau et en le faisant remonter dans le tas binaire jusqu'à ce que la propriété du tas binaire soit respectée. La fonction `insert` ci-dessous illustre l'insertion dans un tas binaire (en supposant que le type `CompleteBinaryTree` est défini de la même manière que dans l'algorithme 3.18) :

```
1  void insert(CompleteBinaryTree *binaryHeap, int value) {
2    binaryHeap->array[binaryHeap->size] = value;
3    binaryHeap->size++;
4    int index = binaryHeap->size - 1;
5    int parentIndex = (index - 1) / 2;
6    while (index > 0 &&
7      binaryHeap->array[index] < binaryHeap->array[parentIndex]) {
8      swap(&binaryHeap->array[index], &binaryHeap->array[parentIndex]);
9      index = parentIndex;
10     parentIndex = (index - 1) / 2;
11   }
12 }
```

Algorithm 3.19 – Insertion dans un tas binaire

La figure 3.10 illustre l'insertion de l'élément 5 dans un tas binaire min.

La valeur 5 est initialement insérée à la fin du tableau (premier fils libre à gauche dans le dernier niveau de l'arbre). Ensuite, elle est comparée à son parent, dans ce cas 8. Comme $5 < 8$, les deux valeurs sont échangées. La valeur 5 est maintenant comparée à son nouveau parent, dans ce cas 3. Comme $5 > 3$, les valeurs ne sont pas échangées et l'insertion est terminée.

3.5.5 Percolation vers le bas

La percolation vers le bas est utilisée pour réorganiser un tas binaire après l'extraction de son minimum. Une fois le minimum extrait, le dernier élément du tableau est placé à la

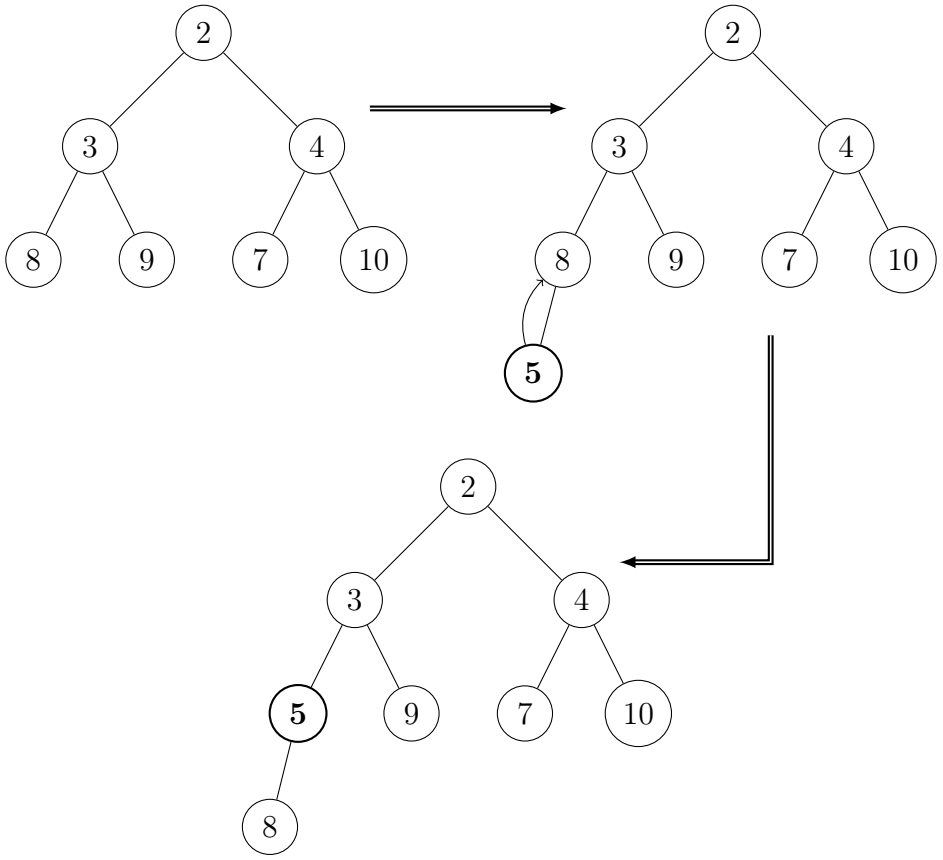

FIGURE 3.10 – Insertion de l'élément 5 dans un tas binaire min

racine de l'arbre. Ensuite, cet élément est déplacé vers le bas jusqu'à ce que la propriété du tas binaire soit respectée. Afin de déplacer l'élément vers le bas, nous le comparons à ses deux enfants et nous échangeons sa valeur avec celle de son plus petit enfant. La fonction `percolateDown` ci-dessous illustre la percolation vers le bas :

```
1  void percolateDown(CompleteBinaryTree *binaryHeap) {
2    int index = 0;
3    int leftChildIndex = 2 * index + 1;
4    int rightChildIndex = 2 * index + 2;
5    while (leftChildIndex < binaryHeap->size) {
6      int minIndex = index;
7      if (binaryHeap->array[leftChildIndex]
8            < binaryHeap->array[minIndex]) {
9        minIndex = leftChildIndex;
10     }
11     if (rightChildIndex < binaryHeap->size &&
12       binaryHeap->array[rightChildIndex] < binaryHeap->array[minIndex]) {
13       minIndex = rightChildIndex;
14     }
```

```
15        if (minIndex != index) {
16          swap(&binaryHeap->array[index], &binaryHeap->array[minIndex]);
17          index = minIndex;
18          leftChildIndex = 2 * index + 1;
19          rightChildIndex = 2 * index + 2;
20        } else {
21          break;
22        }
23      }
24    }
```

Algorithm 3.20 – Percolation vers le bas

Dans la fonction `percolateDown`, nous commençons par comparer la valeur de l'élément à celle de ses deux enfants. Si la valeur de l'élément est supérieure à celle de son plus petit enfant, nous échangeons les deux valeurs. Ensuite, nous répétons le processus jusqu'à ce que la propriété du tas binaire soit respectée ou que l'élément n'ait plus d'enfants.

La figure 3.11 illustre la percolation vers le bas après l'extraction du minimum du même tas binaire présenté dans la figure 3.10.

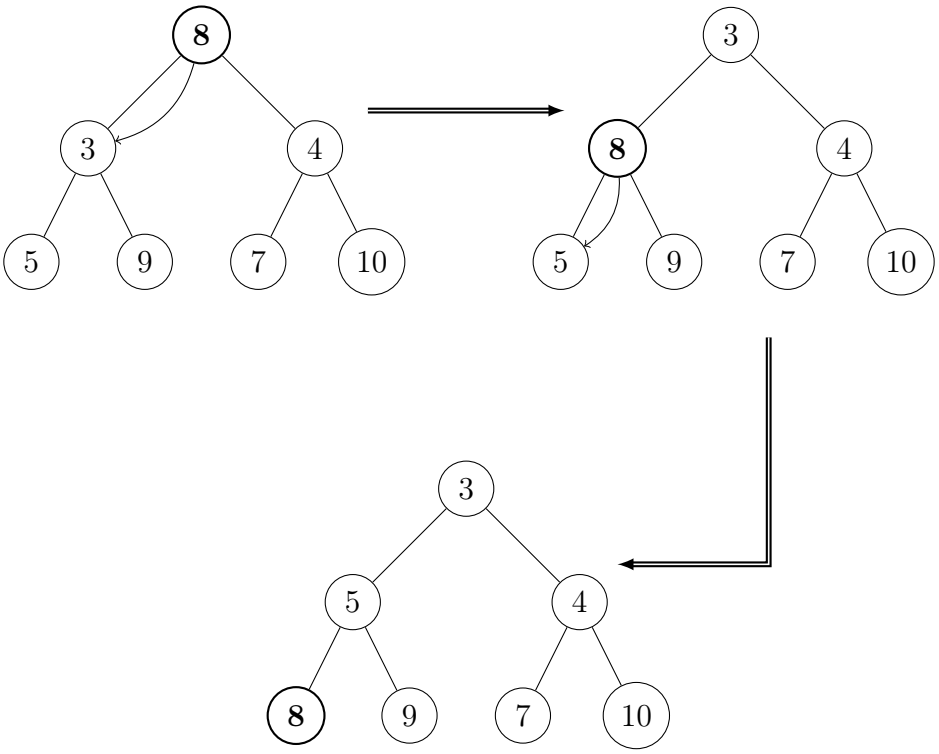

FIGURE 3.11 – Percolation vers le bas après l'extraction du minimum

3.5.6 Algorithmes de tri par tas

Une fois que les processus d'insertion dans un tas binaire et de percolation vers le bas sont bien compris, nous passons à l'implémentation de l'algorithme de tri par tas illustré dans la fonction `heapSort` ci-dessous :

```
1  void heapSort(int *array, int size) {
2    CompleteBinaryTree binaryHeap = initCompleteBinaryTree(size);
3    binaryHeap.array = array;
4    binaryHeap.size = 0;
5    for (int i = 0; i < size; i++) {
6      insert(&binaryHeap, array[i]);
7    }
8    for (int i = 0; i < size; i++) {
9      array[i] = binaryHeap.array[0];
10     binaryHeap.array[0] = binaryHeap.array[binaryHeap.size - 1];
11     binaryHeap.size--;
12     percolateDown(&binaryHeap);
13   }
14 }
```

Algorithm 3.21 – Tri par tas

La fonction `heapSort` suppose l'existance de la fonction `initCompleteBinaryTree` qui crée un tas binaire vide d'une taille maximale passée en paramètre. Les fonctions `insert` et `percolateDown` sont celles présentées dans les sections 3.5.4 et 3.5.5 respectivement. La fonction `heapSort` commence par créer un tas binaire vide de la taille du tableau à trier. Ensuite, elle insère tous les éléments du tableau dans le tas binaire. Après l'insertion de tous les éléments, le tableau est trié. Pour ce faire, la fonction `heapSort` extrait le minimum du tas binaire et le place à la fin du tableau. Ensuite, elle percole le nouvel élément à la racine du tas binaire vers le bas avec la fonction `percolateDown`. Le processus d'extraction du minimum et de percolation vers le bas est répété jusqu'à ce que le tas binaire soit vide.

L'algorithme de tri par tas a une complexité temporelle de $O(n \log n)$ dans le pire des cas. Ses performances sont similaires à celles du tri fusion et du tri rapide.

3.6 Conclusion

Dans ce chapitre, nous avons exploré en détail les structures de données d'arbres, qui jouent un rôle fondamental dans le domaine de l'informatique et de l'algorithmique. Nous avons découvert que les arbres sont des outils polyvalents et puissants pour organiser des données de manière hiérarchique, ce qui les rend indispensables dans une grande variété

de problèmes informatiques. Nous avons commencé par examiner les bases de la structure des arbres et leur terminologie, ensuite, nous avons plongé dans les différentes variantes d'arbres, notamment les arbres binaires, les arbres de recherche binaires et les arbres équilibrés, en comprenant comment ils sont structurés et comment ils peuvent être utilisés pour diverses applications. Nous avons également étudié les principales opérations que nous pouvons effectuer sur les arbres, telles que l'insertion, la suppression, la recherche, le parcours et le tri. Cela nous a permis de comprendre comment manipuler les données dans un arbre et comment accéder efficacement aux informations spécifiques que nous recherchons. Tout au long du chapitre, nous avons également abordé des concepts plus avancés qui permettent d'améliorer encore davantage les performances des opérations clés sur les arbres. Grâce à ces connaissances approfondies, nous sommes désormais prêts à aborder divers problèmes d'algorithmique, tels que la recherche efficace, les algorithmes de tri avancés, les parcours d'arbres et d'autres applications intéressantes.

En conclusion, les arbres sont un pilier essentiel de l'informatique et de l'algorithmique, et la maîtrise de leurs concepts et de leurs opérations est cruciale pour aborder efficacement une grande variété de problèmes. Les compétences acquises dans ce chapitre nous fourniront une base solide pour résoudre des défis algorithmiques complexes et optimiser les performances de nos applications. Nous sommes maintenant prêts à mettre en pratique ces connaissances dans des applications pratiques et à explorer de nouvelles possibilités avec la puissante structure de données qu'est l'arbre.