CC2 du cours d'Algorithmie et Structure de Donnée
Fiche de Révision Approfondie : Arbres, BST, Tas, Graphes et Parcours

GALLAND Romain*

14 décembre 2023

*Merci à ChatGPT

Table des matières

Introduction

Cette fiche de révision couvre en détail les concepts fondamentaux et avancés des arbres, arbres binaires de recherche (BST), tas, graphes et leurs parcours. Chaque section comprend des définitions, des exemples de structures de données en C, des algorithmes et des illustrations pour une compréhension complète.

1 Arbres

1.1 Généralités sur les Arbres

1.1.1 Définitions

Un arbre est une structure de données hiérarchique composée de nœuds, où chaque nœud peut avoir zéro ou plusieurs enfants. Un arbre ne contient pas de cycles, et il y a un seul chemin entre deux nœuds quelconques.

1.1.2 Types d'arbres

- **Arbre binaire** : Chaque nœud a au plus deux enfants.
- **Arbre binaire complet** : Tous les niveaux, sauf éventuellement le dernier, sont entièrement remplis.
- **Arbre binaire équilibré** : La différence de hauteur entre les sous-arbres gauche et droit de tout nœud est au plus un.

1.1.3 Applications des arbres

Les arbres sont utilisés dans de nombreuses applications comme les systèmes de gestion de bases de données, l'analyse syntaxique dans les compilateurs, les algorithmes de recherche, etc.

1.1.4 Terminologie des arbres

- **Racine** : Le nœud supérieur de l'arbre.
- **Feuille** : Un nœud sans enfants.
- **Branche** : Un nœud avec au moins un enfant.
- **Hauteur** : Longueur du chemin le plus long de la racine à une feuille.

1.1.5 Exemples concrets

Un exemple concret d'utilisation des arbres est la représentation des dossiers et fichiers dans un système d'exploitation, où chaque dossier est un nœud et les fichiers/dossiers qu'il contient sont ses enfants.

1.2 Arbres Binaires

1.2.1 Définitions et Propriétés

Un arbre binaire est un type spécial d'arbre dans lequel chaque nœud a au plus deux enfants, souvent désignés comme enfant gauche et enfant droit. Il est utilisé dans de nombreuses applications en informatique en raison de sa structure simple et efficace.

1.2.2 Représentation en C

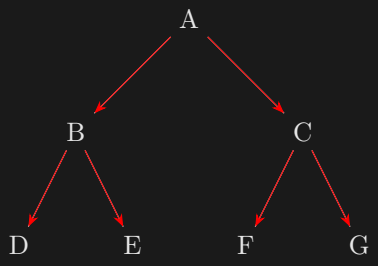
```c
struct struct Node {
    int data;
    struct struct Node* left;
    struct struct Node* right;
};
```

Cette structure en C représente un nœud d'un arbre binaire avec une valeur de données, un pointeur vers le fils gauche et un pointeur vers le fils droit.

1.2.3 Parcours d'arbres binaires

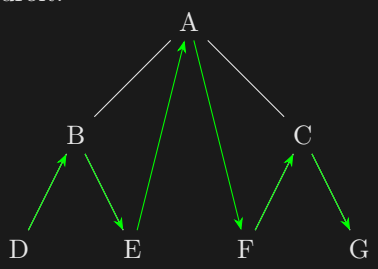
Les arbres binaires peuvent être parcourus de différentes manières :

Parcours préfixe Visite d'abord la racine, puis parcours du sous-arbre gauche, suivi du sous-arbre droit.



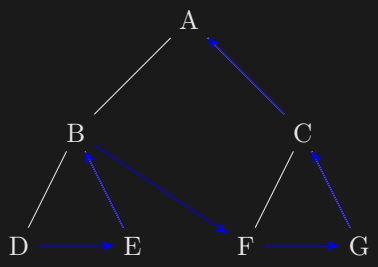
A→B→D→E→C→F→G

Parcours infixe Parcours d'abord le sous-arbre gauche, visite la racine, puis parcours le sous-arbre droit.



D→B→E→A→F→C→G

Parcours postfixe Parcours d'abord le sous-arbre gauche, puis le sous-arbre droit, et enfin visite la racine.



D→E→B→F→G→C→A

1.3 Arbres Binaires de Recherche (BST)

1.3.1 Définitions et Caractéristiques

Un Arbre Binaire de Recherche (BST) est un arbre binaire dans lequel chaque nœud a une clé, et pour chaque nœud, toutes les clés dans le sous-arbre gauche sont inférieures à la clé du nœud, et toutes les clés dans le sous-arbre droit sont supérieures. Cette propriété rend les BST efficaces pour la recherche et l'accès aux données.

1.3.2 Opérations sur les BST

Les opérations principales sur un BST incluent :
- **Insertion** : Ajouter un nouvel élément tout en maintenant la propriété du BST.
- **Recherche** : Trouver un élément dans le BST.
- **Suppression** : Retirer un élément du BST.
- **Parcours** : Parcourir les éléments du BST (préfixe, infixe, postfixe).

1.3.3 Implémentation en C

```c
struct struct Node {
    int key;
    struct struct Node* left;
    struct struct Node* right;
};

struct Node* createNode(int key) {
    struct Node* newNode = (struct Node*) malloc(sizeof(struct Node));
    newNode->key = key;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}
```

Cette structure et fonction en C représentent un nœud de BST et la création d'un nouveau nœud.

1.3.4 Complexité des opérations

	En moyenne	Dans le pire des cas
Insertion	O(log n)	O(n)
Recherche	O(log n)	O(n)
Suppression	O(log n)	O(n)

La complexité dépend de l'équilibre de l'arbre.

1.3.5 Exemples et Applications

- **Systèmes de gestion de bases de données** : Pour des recherches et des accès rapides aux données.
- **Algorithmes de tri** : Le tri par arbre est implémenté en utilisant des BST.
- **Structures de données dynamiques** : Les BST permettent l'insertion et la suppression de données tout en maintenant l'ordre.

1.4 Tas

1.4.1 Définition et Types

Un tas est une structure de données arborescente complète utilisée principalement pour implémenter des files de priorité. Il existe deux types principaux de tas :

- **Tas Max** : La clé à chaque nœud est supérieure aux clés de ses enfants.
- **Tas Min** : La clé à chaque nœud est inférieure aux clés de ses enfants.

1.4.2 Représentation en C

```c
struct Heap {
    int* array;
    int size;
    int capacity;
};
```

Cette structure en C représente un tas avec un tableau pour stocker les éléments, la taille actuelle et la capacité maximale.

1.4.3 Opérations sur les tas

Les principales opérations sur les tas incluent :

- **Insertion** : Ajouter un nouvel élément tout en maintenant la propriété du tas.
- **Suppression** : Retirer l'élément le plus élevé/bas (selon le type de tas).
- **Heapify** : Réorganiser le tas pour maintenir ses propriétés après une opération.

1.4.4 Tri par tas

Le tri par tas est un algorithme de tri efficace utilisant la structure de données du tas. Il consiste à construire un tas à partir des données à trier, puis à déplacer successivement le plus grand élément du tas à la fin du tableau non trié.

1.4.5 Applications des tas

— **Files de priorité** : Implémentation efficace pour gérer des éléments avec des priorités.
— **Tri de données** : Le tri par tas est un algorithme de tri en place avec une bonne complexité moyenne.
— **Algorithmes de graphe** : Comme dans l'algorithme de Dijkstra pour les chemins les plus courts.

2 Graphes

2.1 Généralités sur les Graphes

2.1.1 Définitions et Types

Un graphe est une collection de nœuds (sommets) et d'arêtes qui relient ces nœuds. Il existe deux types principaux de graphes :

- **Graphe non orienté** : Les arêtes n'ont pas de direction.
- **Graphe orienté** : Les arêtes ont une direction définie.

2.1.2 Représentations des graphes

Les graphes peuvent être représentés de plusieurs manières en informatique, notamment :

- **Matrices d'adjacence** : Un tableau 2D représentant les relations entre les nœuds.
- **Listes d'adjacence** : Une liste pour chaque nœud, contenant ses voisins.

2.1.3 Applications des graphes

- **Réseaux sociaux** : Modélisation des relations entre individus.
- **Systèmes de navigation** : Représentation des routes et intersections.
- **Organisation de données** : Comme dans les bases de données relationnelles.

2.1.4 Terminologie des graphes

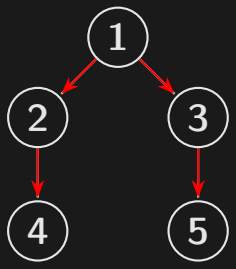
- **Sommets** : Les éléments de base d'un graphe.
- **Arêtes** : Les liens entre les sommets.
- **Voisins** : Les sommets connectés par une arête.
- **Degré** : Le nombre d'arêtes connectées à un sommet.

2.2 Algorithmes sur les Graphes

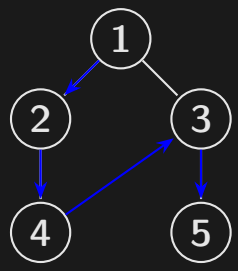
2.2.1 Parcours de graphes (BFS et DFS)

Le parcours de graphes est une méthode fondamentale pour explorer tous les sommets d'un graphe. Il existe deux types principaux de parcours de graphes : le parcours en largeur (BFS) et le parcours en profondeur (DFS).

BFS (Breadth-First Search) Le BFS explore le graphe en largeur. Il commence par le sommet source, puis visite tous les sommets voisins, puis leurs voisins, et ainsi de suite. Il est souvent utilisé pour trouver le chemin le plus court entre deux sommets dans un graphe non pondéré.



DFS (Depth-First Search) Le DFS explore le graphe en profondeur. Il commence par le sommet source, puis visite un sommet voisin, puis un voisin de ce voisin, et ainsi de suite, jusqu'à ce qu'il atteigne un sommet sans voisins non visités, puis il revient en arrière. Le DFS est souvent utilisé pour vérifier l'existence d'un chemin entre deux sommets.



2.2.2 Algorithmes de chemins les plus courts

Les algorithmes de chemins les plus courts sont utilisés pour trouver le chemin le plus court entre deux sommets dans un graphe. Les deux algorithmes les plus connus sont l'algorithme de Dijkstra et l'algorithme de Bellman-Ford.

Algorithme de Dijkstra L'algorithme de Dijkstra est utilisé pour trouver le chemin le plus court entre un sommet source et tous les autres sommets dans un graphe pondéré avec des poids positifs.

Algorithme de Bellman-Ford L'algorithme de Bellman-Ford est utilisé pour trouver le chemin le plus court dans un graphe pondéré, même avec des poids négatifs. Cependant, il ne peut pas gérer les cycles de poids négatif.

2.2.3 Arbres couvrants minimums

Un arbre couvrant minimum (ACM) d'un graphe est un sous-graphe qui est un arbre, qui contient tous les sommets du graphe, et dont le poids total (la somme des poids de toutes les arêtes) est minimal. Les deux algorithmes les plus connus pour trouver un ACM sont l'algorithme de Prim et l'algorithme de Kruskal.

2.2.4 Applications algorithmiques

Les algorithmes sur les graphes sont utilisés dans de nombreux domaines, tels que les réseaux de télécommunication, les systèmes de navigation, l'optimisation de réseaux, la planification d'itinéraires, la conception de circuits intégrés, et bien d'autres.

3 Comparaison des Structures de Données : Arbres, BST et Tas

3.1 Introduction

Cette section compare les arbres, les arbres binaires de recherche (BST) et les tas, en mettant en évidence leurs caractéristiques, avantages et cas d'utilisation optimaux.

3.2 Arbres

Caractéristiques : Les arbres sont des structures hiérarchiques avec des nœuds et des enfants. Chaque nœud peut avoir plusieurs enfants, et il n'existe pas de contrainte spécifique sur le nombre d'enfants par nœud.

Utilisation : Idéal pour représenter des structures naturellement hiérarchiques, comme des systèmes de fichiers, des arbres de décision, ou des structures organisationnelles.

3.3 Arbres Binaires de Recherche (BST)

Caractéristiques : Un BST est un type d'arbre binaire où chaque nœud a une clé, et toutes les clés dans le sous-arbre gauche d'un nœud sont inférieures à la clé du nœud, et celles dans le sous-arbre droit sont supérieures.

Utilisation : Excellent pour des opérations de recherche, d'insertion et de suppression efficaces. Utilisé dans des situations où les données doivent être dynamiquement insérées ou supprimées tout en maintenant un ordre.

3.4 Tas

Caractéristiques : Un tas est un arbre binaire complet, organisé de telle manière que la valeur de chaque nœud est supérieure (dans un tas max) ou inférieure (dans un tas min) à celle de ses enfants.

Utilisation : Idéal pour des files de priorité où l'accès à l'élément le plus grand ou le plus petit est fréquent. Utilisé dans les algorithmes de tri par tas et dans des algorithmes de graphes comme l'algorithme de Dijkstra.

3.5 Choix de la Structure de Données

— **Arbres** : Choisissez cette structure lorsque vous avez besoin de représenter des données hiérarchiques avec des relations parent-enfant complexes.
— **BST** : Privilégiez les BST pour des collections dynamiques de données où les opérations de recherche, d'insertion et de suppression sont fréquentes et doivent être rapides.
— **Tas** : Utilisez les tas pour les applications nécessitant un accès rapide au plus grand ou au plus petit élément, comme les files de priorité ou les algorithmes de tri.