Lecture Notes for Data Structures

Emulie Chhor

May 15, 2021

1 Overview of the field of Data Structures and Algorithms

One of the most important field of computer science is about Data Structures and algorithms, which is learn via 4 courses:

1. Discrete Maths
2. Data Structures
3. Algorithms
4. Theory of Computation
5. Modèle de Recherche Opérationnelle (optional)

Remarque Problème Intuition

2 Why study Data Structures

3 How to study Data Structures

Lorsqu'on étudie les structures de données, on doit comprendre que chaque structure de donnée possède un invariant qui lui donne ses propriétés, et on doit préserver ces propriétés lors des opérations:

1. Propriétés
2. Invariant
3. Helper Methods
4. Opérations: insertion, suppresion, recherche
5. Complexité des opérations: worse, best, average
6. Implementation

4 Introduction

The study of data structures can be divided in the following chapters:

1. Basics Data Structures: Dynamic Arrays and Linked List
2. Basics ADT: Bag, Stack and Queue
3. Sorting Algorithms
4. Heap
5. Treap (optional)
6. Binary Search Trees
7. Splay (optional)
8. Balanced Binary Trees
9. Hash Table
10. Graph Theory

5 Basic Data Structures

5.1 Overview

Le premier chapitre en structure des données introduit les notions de dynamic array et de linked list. En gros, ce sont des structures de données qui sont utilisées pour implémenter des structures abstraites.

Ces structures de données supportent certaines opérations

1. insertion
2. deletion
3. recherche

5.2 Dynamic Arrays

Le dynamic array est utilisé surtout pour storer des éléments dont on connait la taille et l'index parce qu'il requiert d'être resized

1. insertion: si la capacité est atteinte, on double la capacité et on copie les données dans le nouvel array
2. deletion: si la capacité est le quart de la taille du array, on copie les éléments dans un array qui est la moitié de la taille initiale

3. recherche: recherche linéaire si éléments non-ordonnées, recherche binaire si les éléments sont ordonnées

Complexité du resize: Preuve Crédit-Débit

On peut montrer que le resize prend un temps constant amorti par une preuve crédit-débit. Soit 1$, le cout pour push et pop. On sait qu'à chaque puissance de n éléments, on va devoir insérer n éléments et resize 1 fois. Ainsi, on va devoir:

1. ajouter les éléments une première fois (on n'a pas atteint la capacité): n fois
2. resize: coute 2n, car on doit pop n éléments et push n éléments dans le nouvel array
3. nombre d'éléments: n

Ainsi, le cout amorti est de $\frac{n+2n}{n} = 3$

5.3 Linked List

Il existe 3 types de linked list:

1. Singly Linked List: Peuvent seulement être traversée du head au tail, car on n'a pas de pointeur au précédant
2. Doubly Linked List: Can be traversed backward because we store the previous node, but takes twice the space
3. Circular Linked List: the head and tail can reach each other

Le linked list supporte les opérations suivantes:

1. insertion: On traverse la liste séquentiellement, et on ajoute le nouveau noeud entre le previous et le prochain en sauvegardant le pointeur du prochain dans une variable temporaire
2. deletion: On change le pointeur du previous et du prochain en sauvegardant le pointeur du prochain avant d'enlever le noeud
3. recherche: On doit faire de la recherche séquentielle parce qu'on n'a pas accès aux indices

5.4 Dynamic Arrays vs Linked List

6 Basic Abstract Data Types

6.1 Overview

Le bag, stack et queue sont les DS les plus basiques. On peut les implémenter avec des arrays, mais il est préférable de le faire avec des arraylist, puisqu'on ne connait pas leur taille d'avance (et on ne travaille qu'avec le head/tail)

6.2 Bag

Le bag ne supporte qu'une seule opérations: insertion. Pour la recherche, on ne peut faire qu'une recherche séquentielle, car les éléments n'ont pas d'ordre.

6.3 Stack

Le stack, communément appelé FILO (first-in, last-out), est un type abstrait qui est utilisé pour le cache et l'OS. Elle supporte les 3 opérations de base:

1. push (insertion): ajoute le nouvel élément au top
2. pop (suppression): si non vide, enleve l'élément eu top
3. recherche: pas fait pour ça, mais on peut peek

6.4 Queue

La queue, communément appelée FIFO (first-in, first-out), supporte les 3 opérations de base

1. enqueue (insertion): insère l'élément à la fin
2. dequeue (suppression): si non vide, enlève l'élément en tête de la liste
3. recherche: pas fait pour ça, mais on peut peek

7 Sorting Algorithms

7.1 Overview

Idéalement, le meilleur algorithme de tri devrait être linéarithmique et ne pas utiliser d'espace auxiliaire. Cependant, il n'existe pas de sorting algorithms satisfaisant ces 2 propriétés. Ainsi, on peut discerner quelques sorting algorithms:

1. Selection Sort
2. Insertion Sort
3. Bubble Sort
4. Mergesort
5. Quicksort
6. Heapsort
7. Raddix Sort

Notons qu'on apprend les sorting algorithms, non pas pour les implémenter, mais plutôt parce que leur structure et temps de complexité sont de bons introductions à l'analyse de la complexité. On veut donc être en mesure de connaitre le input qui va donner un temps optimal, temps moyen, temps pire temps.

7.2 Selection Sort

7.3 Insertion Sort

7.4 Bubble Sort

7.5 Mergesort

7.6 Quicksort

7.7 Heapsort

7.8 Raddix Sort

8 Heap

8.1 Invariant

Un heap est une structure de données qui implémente des files de propriété. Il existe 2 types de heap:

1. MinHeap: $parent < child$
2. MaxHeap: $parent > child$

Remarquons qu'on a des inégalités strictes, car on ne veut pas ajouter les mêmes valeurs au heap. De plus, l'implémentation est similaire, la seule chose qui change est la comparaison qu'on fait: less vs more

8.2 Helper Methods

Le heap possède 2 helpers methods: swim et sink

Swim

1. Trouver le parent
2. Échanger avec le parent jusqu'à l'invariant soit satisfait ou qu'on aille atteint le root

Sink

1. Trouver le plus grand/petit enfant
2. Échanger avec l'enfant jusqu'à l'invariant soit satisfait ou qu'on atteigne un NIL

8.3 Operations

1. Insertion
    - ajouter l'élément à la dernière position
    - swim() pour retrouver sa bonne position
2. Suppression
    - Échanger le root avec le dernier élément
    - supprimer le dernier élément
    - sink le root pour qu'il soit à la bonne position
3. Recherche: Puisqu'il n'y a pas de relations entre les 2 enfants, on doit faire une recherche linéaire

8.4 Complexité des Opérations

9 Binary Search Trees

9.1 Overview

Le BST est une structure de donnée utilisée, car elle maintient un ordre. Son invariant: $node.left < node < node.right$. Comme avec le heap, on n'accepte pas de valeurs duppliquées

9.2 Helper Methods

Les helpers methods sont:

1. FindMin: trouver la plus petite valeur dans le sous-arbre en parcourant l'arbre en allant toujours vers la gauche
2. FindMax: trouver la plus grande valeur dans le sous-arbre en parcourant l'arbre en allant toujours vers la droite

Ces helpers methods sont utiles, car elles nous permet de choisir le successeur/prédécesseur lors de la suppression

9.3 Operations

1. Insertion: Parcourir l'arbre
    - Si element ¡ node.val: allez à gauche
    - Si element ¿ node.val: allez à droite
    - Si element = node.val: ne pas ajouter de duppliqué
2. Suppression

- Parcourir le tree pour trouver l'élément à enlever
- Choisir le prédécesseur/successeur si le noeud supprimé possède un enfant (ou plus)

3. Recherhe: Puisqu'on maintient une structure, on peut effectuer une recherche binaire

9.4 Complexité

9.5 Implementation

Pour l'implémentation, on déclare une classe noeud qui a les paramètres suivants:

1. value
2. left child
3. right child
4. (optional) parent
5. (optional) height

9.5.1 Operations in Java

TODO

10 Balanced Binary Search Trees

10.1 Overview

Les balanced Binary Trees sont la version 2.0 des BST. Essentiellement, on a vu que les arbres BST ne garantissent pas une hauteur logarithmique, alors le temps de complexité de ses opérations peut aller jusqu'à O(n), ce qui n'est pas bon. Les BBST sont des arbres, qui, par leurs invariants, permettent de garantir une hauteur logarithmique et ainsi un temps de complexité O(log n) au pire.

On discerne 3 BBST:

1. AVL Trees: rotations et balance factor
2. Red-Black Trees: rotation et couleur
3. B-Trees: noeuds avec plusieurs éléments

Notons que la recherche pour le AVL et le Red-Black Tree se font avec un binary search, car ce sont des BST balancés.

10.2 AVL Trees

10.2.1 Overview

Les arbres AVL qui utilisent la rotation et un balance factor pour garantir un temps logarithmique au pire. Pour ce faire, on doit respecter l'invariant suivant: la différence de hauteur entre 2 sous-arbres doit être d'au plus 1.

10.2.2 Helper Methods: Rotations

1. Rotations: left and right
2. FindHeight: Calculer balance factor

Rotations

Pour un left rotation, on a un arbre right heavy et on veut le rendre left heavy:

1. right child becomes the root
2. root becomes left child of new root
3. rigt child.left becomes right child of old root

De la même façon, pour un right rotation, on a un arbre left heavy et on veut le rendre right heavy

FindHeight

On veut trouver la hauteur du subtree, car cela nous permet de calculer le balance factor.

$$return 1 + max(findHeight(node.left), findHeight(node.right))$$

10.2.3 Operations

Operations

1. Insertion
   - Perform BST insertion
   - Update balance factor
   - Perform Rotation if invariant violated
2. Suppression
   - Perform BST deletion
   - Update balance factor
   - Perform Rotation if invariant violated

Updating Balance Factor

1. recalculer hauteur à partir du root

[Balance Factor]

$$returnheight(x.left) - height(x.right)$$

Balance an AVL Tree
Lorsque l'invariant est violé (bf ¿1), on doit rotater les branches. En considérant que zig: left heavy et zag: right heavy. On doit considérer 4 cas:

1. zig-zig: rotateRight
2. zag-zag: rotateRight
3. zig-zag: rotateLeft pour avoir zig-zig
4. zag-zig: rotateRight pour avoir zag-zag

Associer le balance factor à chaque cas
TODO

10.2.4 Implementation

Princeton Implementation: `https://algs4.cs.princeton.edu/code/edu/princeton/cs/algs4/AVLTreeST.java.html`

10.3 Red-Black Trees

10.3.1 Overview

Les red-black trees utilisent la rotation et la couleur pour garantir une hauteur logarithmique. L'invariant est donné p/r à la hauteur noire de l'arbre: les noeud de couleur rouge = 0, alors que les noeuds de couleur noire comptent pour 1. Les red0black trees respectent les propriétés suivantes:

1. Node is red at insertion
2. Root node and NIL nodes are black
3. Child of red nodes are black
4. Black height from root to NIL is the same for all paths

Ils possèdent les caractéristiques suivantes:

1. Shortest Path: alternating red-black trees
2. Longest Path: only black node

10.3.2 Operations

1. Insertion
   -
2. Suppression

10.3.3 Implementation

Princeton Red-Black Tree Implementation: `https://algs4.cs.princeton.edu/33balanced/RedBlackBST.java.html`

10.4 B-Trees

11 Hash Table

11.1 Collisions

11.2 Load Factor

11.3 Dealing with Collision

11.3.1 Separate Chaining

11.3.2 Open Adressing

Linear Probing Double Hashing

12 Graph Theory

12.1 Terminology

12.2 Graph Representation

1. Edges List
2. Adjacency List
3. Adjacency Matrix

12.3 Problems in Graph Theory