TD sur le chapitre 2 : types de données abstraits

Partie 1 : Tableaux

Exercice 1.1 : On considère un tableau non trié de *n* éléments.

- Ecrivez un algorithme qui recherche un élément dans ce tableau.
- Prouvez cet algorithme.
- Quelle est la complexité de cet algorithme ?

Exercice 1.2 : On considère un tableau trié de *n* éléments.

- Ecrivez un algorithme qui recherche un élément dans ce tableau, par dichotomie.
- Prouvez cet algorithme.
- Quelle est la complexité de cet algorithme ? Vous pouvez utiliser la formule de Master.

Exercice 1.3 : L'algorithme de sélection du maximum permet de trier un tableau. Pour cela, il commence par trouver le plus grand élément du tableau, et le place à la fin du tableau. Puis, il recommence sur un tableau sans le dernier élément. L'algorithme continue ainsi jusqu'à ce qu'il ne reste plus qu'un seul élément dans le tableau.

- Ecrivez cet algorithme.
- Prouvez cet algorithme.
- Quelle est la complexité de cet algorithme ?

Partie 2 : Listes

Exercice 2.1 : Considérez les fonctions suivantes : creerListe, afficherListe, rechercheDansListe, ajoutTeteListe, ajoutFinListe. Implémentez chacune de ces fonctions dans le cas :

- d'une liste simplement chaînée,
- d'une liste doublement chaînée,
- d'une liste simplement chaînée circulaire,
- d'une liste doublement chaînée circulaire.

Calculez la complexité de chacune de ces opérations.

Partie 3 : Piles

Exercice 3.1 : Implémentez une pile avec un tableau.

Exercice 3.2 : Implémentez une pile avec une liste simplement chaînée.

Exercice 3.3 : En utilisant une pile, écrivez un algorithme qui inverse les éléments d'une liste (c'est-à-dire, cet algorithme renvoie [3,2,1] quand on lui donne la liste [1,2,3]).

Partie 4 : Files

Exercice 4.1 : Implémentez une file avec un tableau circulaire.

Exercice 4.2 : Implémentez une file avec une liste, avec ajout en fin.

Exercice 4.3 : Implémentez une file avec une liste doublement chaînée, avec ajout en tête.

Partie 5 : Arbres

Exercice 5.1 : Dessinez un arbre représentant l'expression arithmétique suivante : (3*1)+(4*(2+6)).

Exercice 5.2 : Ecrivez un algorithme qui affiche toutes les valeurs d'un arbre binaire.

Exercice 5.3 : Ecrivez un algorithme qui calcule la hauteur d'un arbre binaire.

Exercice 5.4 : Un arbre binaire de recherche est un arbre binaire dans lequel pour chaque noeud, la valeur du noeud est strictement supérieure à la valeur de tous les noeuds du fils gauche, et inférieure ou égale à la valeur de tous les noeuds du fils droit. Ecrivez un algorithme qui vérifie si un arbre donné en paramètre est un arbre binaire de recherche.

Exercice 5.5 : Ecrivez un algorithme qui prend un arbre et deux noeuds en paramètre, et qui calcule le plus proche ancêtre commun aux deux noeuds.

Partie 6 : Tas

Pour obtenir les complexités souhaitées du tas, on a besoin d'une structure de données assez complexe. On va ainsi considérer qu'un tas est un arbre binaire complet (c'est-à-dire que tous les niveaux de l'arbre sont pleins, à l'exception éventuellement du dernier niveau) avec les contraintes suivantes :

- Sur le dernier niveau, les noeuds sont remplis de gauche à droite.
- A chaque noeud de l'arbre, l'élément est plus petit que tous les éléments des descendants.

Exercice 6.1 : Ecrivez un algorithme en $O(1)$ qui retourne le plus petit noeud du tas.

Exercice 6.2 : Montrez que la hauteur d'un tas de n éléments est en $O(log(n))$.

Exercice 6.3 : Pour ajouter un élément dans un tas, on commence par l'ajouter au dernier niveau, sur la seule position possible (donc sur le noeud libre le plus à gauche du dernier niveau). Puis, essaye de « corriger » ce tas. Pour cela, l'algorithme compare le nouveau noeud avec son père, et les échange si le noeud est plus petit. L'algorithme continue ensuite de remonter jusqu'à la racine de l'arbre. Ecrivez un algorithme qui réalise cette opération.

Exercice 6.4 : Pour supprimer un élément d'un tas, on retire toujours la racine. On prend alors la dernière feuille de l'arbre, que l'on met à la racine, et on la fait redescendre niveau par niveau. Ecrivez un algorithme qui réalise cette opération.

Partie 7 – Exemples concrets

Partie 7.1 – Algorithme Union-Find

Le TDA Union-Find représente une partition d'un ensemble fini. Elle dispose de trois opérations :

- MakeSet(*x*) : crée une partition pour l'élément *x*.
- Find(*x*) : détermine la partition dans laquelle se trouve *x*.
- Union(*x*,*y*) : fusionne les partitions de *x* et de *y* en une unique partition.

On considère une première version de ce TDA, dans laquelle chaque partition est représentée par un arbre, et la racine de l'arbre contenant un noeud *x* est le représentant de la partition. Pour représenter les arbres, on ne mémorise que le pointeur d'un noeud vers son père. On utilise alors un tableau de *n* éléments, contenant pour chaque noeud uniquement son père.

Exercice 7.1.1 : Ecrivez la fonction MakeSet(x), qui se contente de fixer le père de *x* à *NULL*.

Exercice 7.1.2 : Ecrivez la fonction Find(*x*), qui retourne la racine de l'arbre dans lequel *x* se trouve.

Exercice 7.1.3 : Ecrivez la fonction Union(*x*,*y*), qui cherche les racines de x et de y, et affecte (arbitrairement) *x* comme père de *y*.

Exercice 7.1.4 : Pourquoi ces algorithmes ne permettent pas d'obtenir une bonne complexité ?

On considère une deuxième version de ce TDA, avec deux modifications :

- On ajoute une information appelée « rang » à chaque noeud. Initialement, les rangs sont tous de 0. Quand on fusionne deux arbres, on attache celui qui a le plus petit rang à celui qui a le plus grand rang. En cas d'égalité des rangs, on fait un attachement arbitraire, mais on augmente de un le rang final.
- A chaque fois que la fonction Find(*x*) parcourt un noeud, on change le père de ce noeud par la racine de l'arbre.

Exercice 7.1.5 : A quoi sert la première modification ?

Exercice 7.1.6 : A quoi sert la deuxième modification ?

Exercice 7.1.7 : Programmez les trois fonctions MakeSet(*x*), Find(*x*) et Union(*x*,*y*) avec ces deux modifications.

Remarque : Cette nouvelle version permet d'atteindre une excellente complexité (amortie) de *O(alpha(n))*, avec *alpha(n)* une fonction qui croît extrêmement lentement. En pratique, *alpha(n)<=5* pour toute valeur de *n* (qui n'est pas démesurément grande).

Partie 7.2 – Algorithme de Prim

Soit *G* un graphe non orienté pondéré. Un arbre couvrant de poids minimum est un arbre qui couvre tous les noeuds de *G*, et dont le poids est minimum.

L'algorithme de Prim est un algorithme pour calculer un arbre couvrant de poids minimum. Il fonctionne ainsi :

- On part d'un noeud arbitraire, et on crée un arbre T qui ne contient que ce noeud.
- Tant qu'il reste un noeud de G non couvert par T, on cherche l'arête (x,y) de poids minimal, telle que x appartient à T et y n'appartient pas à T. On ajoute (x,y) à T.

Exercice 7.2.1 : Ecrivez cet algorithme en considérant :

- que le graphe est représenté par une liste d'adjacence (c'est-à-dire, par une liste de toutes les arêtes),
- que les arêtes du graphe sont toutes parcourues à chaque itération.

Exercice 7.2.2 : Quelle est la complexité de cet algorithme ?

Exercice 7.2.3 : Ecrivez cet algorithme en considérant :

- que le graphe est représenté par une liste d'adjacence (c'est-à-dire, par une liste de toutes les arêtes),
- que les arêtes qui sont couvertes par au moins un noeud de T sont ajoutées à un tas.

Exercice 7.2.4 : Quelle est la complexité de cet algorithme ?

Partie 7.3 – Algorithme de Kruskal

L'algorithme de Kruskal est un autre algorithme pour calculer un arbre couvrant de poids minimum. Il fonctionne ainsi :

- On génère une liste d'arbres, un par noeud du graphe, et chaque arbre n'ayant que ce noeud.
- Tant qu'il reste au moins deux arbres, on cherche les deux arbres les plus proches et on les fusionne.

Exercice 7.3.1 : Ecrivez cet algorithme, en utilisant une liste des arêtes, triée par ordre croissant, et la structure Union-Find pour savoir à quel arbre chaque arête appartient.

Exercice 7.3.2 : Quelle est la complexité de cet algorithme ?