CONTRÔLE TERMINAL 2023
(DEMI-)UE STRUCTURES DE DONNÉES AVANCÉES

Aucun document **n'**est autorisé pour cet examen.

Les fonctions doivent être écrites en C, sauf pour le problème.

Questions théoriques (4.5 points)

Q1. Quelle est la complexité asymptotique de l'opération `unir` dans la structure forêt d'Union Find ? (0.75 point)

Q2. Modifions le fonctionnement classique de la structure *tableau dynamique* de sorte qu'à chaque réallocation, la capacité ne soit pas plus doublée, mais qu'elle soit augmentée de 2.

Quelle est alors la complexité amortie d'un ajout d'un élément ? Justifiez votre réponse par un calcul. (3 points)

Q3. Pourquoi dans notre structure de *table de hachage* vue en cours on a imposé la condition que le nombre d'alvéoles est inférieur à 8 fois le nombre d'éléments ? (0.75 point)

Questions pratiques (11.5 pts)

J'ai écrit un petit générateur aléatoire de sujet avec les fonctions vues en TP. Pour montrer que je n'ai pas triché, voici son résultat :

```
Python 3.8.10 (default, Mar 13 2023, 10:26:41)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: /home/courtiel/master/structures-de-donnees-avancees/CT/generateur_sujet.py
>>>
>>> genere_sujet()
[('ensemble', 'supprimer(sans_contrainte_sur_les_alveoles)', 14), ('unionfind', 'initialiser_partition', 10), ('file
_priorite', 'liste_vers_file_priorite', 15), ('avl', 'hauteur_avl', 12)]
>>>
```

EXERCICE 1 – Suppression d'un élément dans un ensemble (3.5 points)

Q1. Écrire la fonction `supprimer_lc` dont le prototype se décrit comme ci-dessous :

```
/** @brief Supprime une occurrence d'une valeur dans une liste chaînée.
 * La mémoire associée au noeud supprimé est libérée.
 * Les autres noeuds gardent la même adresse (aucun nouveau noeud n'est créé).
 * Si la valeur n'est pas dans la liste chaînée, affiche un message d'erreur.
 * **Complexité :** O(taille de la liste)
 * @param l liste chaînée,
 * @param x valeur dont on souhaite supprimer une occurrence dans l.
 * @returns l dans laquelle on a supprimé un noeud de valeur x (s'il existe).
 */
ListeChainee supprimer_lc(ListeChainee l, type_base x);
```

Pour rappel, le type `ListeChainee` se définit comme suit :

```
struct Noeud{
        type_base valeur; /**< L'étiquette du noeud */
        struct Noeud* suivant; /**< L'adresse du noeud suivant.
        Si le noeud est le dernier de la liste,
        alors ce champ est le pointeur nul. */
};

typedef struct Noeud* ListeChainee;
```

Q2. Écrire la fonction `supprimer_sans_reallouer` dont le prototype se décrit comme ci-dessous :

```
/**
 * @brief Supprime une occurrence d'un élément dans la table de hachage. \n
 * On NE réallouera PAS de nouvelle table de hachage
 * même si le nombre d'alvéoles est supérieur à un huitième
 * du nombre d'éléments.
 * Affiche un message d'erreur si jamais la valeur n'est pas dans e. \n
 * **Complexité :** O(1) (en moyenne et en amorti)
 * @param e un ensemble,
 * @param x une valeur dont on veut supprimer une occurrence dans e.
 */
void supprimer_sans_reallouer(Ensemble e, type_base x);
```

Contrairement en TP, on ne préoccupera pas de déplacer les éléments vers une table de hachage plus petite même si le nombre d'alvéoles est trop grand.

Pour rappel, le type `Ensemble` se définit comme suit :

```
struct TableHachage{

        ListeChainee* table; /**< L'adresse de la table de hachage. */
        size_t nb_alveoles; /**< La taille du tableau **table** */
        size_t taille;  /**< Le nombre d'éléments dans la table. */
        double A; /**< La constante A dans la fonction de hachage */
};

typedef struct TableHachage* Ensemble;
```

Pour écrire le fonction `supprimer_sans_reallouer`, on pourra utiliser **sans la rééccrire** la fonction `alveole` dont le prototype est donné par :

```
/**
 * @brief Donne le numéro de l'alvéole où est censée
 * se trouver une certaine valeur.
 * **Complexité :** O(1)
 * @param e un ensemble,
 * @param x une valeur,
 * @returns le code de hachage de x pour e.
 */
size_t alveole(Ensemble e, type_base x);
```

EXERCICE 2 – Initialisation d'une partition vide (Union Find – 2 pts)

Écrire la fonction `initialiser_partition` avec pour prototype :

```
/** @brief Crée une partition faite uniquement avec des singletons
 * (= ensembles de taille 1)
 * **Complexité :** O(nombre_elements)
 * @param nombre_elements le nombre d'éléments de base.
 * @returns une partition de taille 'nombre_elements'
 *  où chaque élément est de taille 1.
 */
Partition initialiser_partition(size_t nombre_elements);
```

Pour rappel :

```
struct UnionFind {

        size_t nombre_elements; /**<  Le nombre d'éléments de base. */
        size_t nombre_ensembles;  /**< Le nombre d'ensembles. */
        Liste parent; /**<  'parent[i]' indique le numéro du parent de 'i'.
        * On rappelle que 'i' est une racine d'un arbre de la forêt
        * si et seulement si parent[i] vaut i. */
        Liste taille_arbre; /**< Si 'i' est racine d'un arbre de la forêt,
        * alors 'taille_arbre[i]' est le nombre d'éléments dans cet arbre. */
};

typedef struct UnionFind* Partition;
```

On pourra évidemment utiliser les fonctions sur les listes (sans les réécrire). Une appendice en fin du sujet liste les prototypes des fonctions sur les listes.

EXERCICE 3 – Liste vers file de priorité (3.5 pts)

Écrire la fonction `liste_vers_file_priorite_min` avec pour prototype :

```
/**
 * @brief Convertit une liste d'entiers en une file de priorité min.
 * **Complexité :** O(n), où n est la taille de la liste.
 * @param l une liste,
 * @returns une file de priorité **minimum** contenant les mêmes éléments que l
 */
FilePriorite liste_vers_file_priorite_min(Liste l);
```

On utilisera la structure :

```
struct TasBinaire {

        Liste valeurs;   /**< Une liste qui va contenir les différentes valeurs*/

};

typedef struct TasBinaire* FilePriorite;
```

Le code sera simplifié par rapport au vrai TP, dans la mesure où :

- plus besoin de traîner une fonction attribut `est_plus_petit`, on utilisera le signe < vu qu'on travaille avec des entiers.
- On transformera forcément la liste en une file de priorité min (en TP, on se laissait aussi la possibilité de la transformer en file de priorité max).

On pourra utiliser les fonctions sur les listes (voir appendice) et les fonctions (et seulement celles-ci – si vous en avez besoin d'autres, il faudra les réécrire) :

```
/** @brief Renvoie une file de priorité vide.
 * **Complexité :** O(1)
 * @returns une file de priorité sans éléments.*/
FilePriorite file_priorite_vide();


/** @brief Renvoie la position de l'enfant gauche.
 * **Complexité :** O(1)
 * @param pos un index dans le champ 'valeurs' d'une file de priorité.
 * @returns l'index de l'enfant qui se situe à gauche de l'élément */
size_t position_enfant_gauche(size_t pos);


/** @brief Renvoie la position de l'enfant droit.\n
 * **Complexité :** O(1)
 * @param pos un index dans le champ 'valeurs' d'une file de priorité.
 * @returns l'index de l'enfant qui se situe à droite de l'élément */
size_t position_enfant_droit(size_t pos);


/** @brief Renvoie la position du parent.\n
 * Provoque une erreur si jamais l'argument vaut 0. \n
 * **Complexité :** O(1)
 * @param pos un index dans le champ 'valeurs' d'une file de priorité.
 * @returns l'index du parent  */
size_t position_parent(size_t pos);
```

EXERCICE 4 – Hauteur d'un AVL (2.5 pts)

Écrire la fonction `hauteur_avl` avec pour prototype :

```
/**
 * @brief Renvoie la hauteur d'un avl. \n
 * **Complexité :** O(log(nombre de noeuds)) –
 * pas besoin ici de parcourir tout l'arbre !
 * @param a un avl.
 * @returns la hauteur de a.
 */
int hauteur_avl(avl a);
```

Pour rappel :

```
struct NoeudAvl {

        int valeur; /**< La valeur stockée dans le noeud.
        * (Ici pas de couples clé/valeur pour simplifier) */

        int facteur_equilibrage; /**< La différence entre la hauteur
        * du sous-arbre droit avec le sous-arbre gauche.
        * Si le facteur d'équilibrage est positif, alors l'arbre penche à
        * droite ; sinon, il penche à gauche. */

        struct NoeudAvl* gauche; /**< Le pointeur vers le sous-arbre gauche.
        Vaut NULL si le noeud n'a pas d'enfant gauche. */

        struct NoeudAvl* droite; /**< Le pointeur vers le sous-arbre droit.
        Vaut NULL si le noeud n'a pas d'enfant droit. */
};

typedef struct NoeudAvl* avl;
```

Problème (4 pts)

EXERCICE 5 – Cassage de cailloux

On vous donne une liste d'entiers `cailloux` tels que `cailloux[i]` est le poids du `i`-ième caillou.

Ensuite, vous participez à un jeu (qui défoule). Chaque tour, vous prenez les **deux** cailloux **les plus lourds** et vous les fracassez l'un contre l'autre.

Supposons que les poids des deux plus gros cailloux sont x et y avec $x \leq y$:

Si $x = y$, les deux cailloux sont détruits.

Si $x < y$, seul le deuxième caillou le plus gros est détruit et le caillou le plus lourd a maintenant un poids de $y - x$.

A la fin du jeu, il reste 0 ou 1 caillou.

Le problème est de déterminer le poids du dernier caillou, s'il existe. Si au contraire, il n'y a plus de cailloux à la fin, on renverra `0`.

Par exemple, si la liste de cailloux est `[6,5,10,7]`, on prend d'abord les cailloux de poids `10` et `7`, on les fracasse : il reste un caillou de poids `3`. Puis, on prend ceux de poids `6` et `5`, on obtient un caillou de poids `1`. Les deux cailloux restants ont un poids `3` et `1` : on obtient un caillou de poids `2`. C'est `2` qu'il faut donc renvoyer.

Q1. Quelle structure de données vu en cours utiliseriez-vous pour ce problème ?

Q2. Écrire **en pseudo-code** un algorithme qui résout ce problème.

Q3. Quelle est la complexité de votre algorithme ?

Appendice : fonctions sur les listes

```
/**
 * @brief La taille de la liste.
 * @param l liste.
 * @returns le nombre d'éléments dans la liste.
 */
size_t longueur(Liste l);

/**
 * @brief Renvoie une liste sans élément.
 * Par défaut, la capacité est fixée à 8 et on double la capacité à
 * chaque réallocation.
 * Complexité : O(1)
 * @returns une liste vide. */
Liste liste_vide();

/**
 * @brief Désalloue la mémoire associée à la liste
 * Désalloue le tableau et la structure elle-même.
 * Complexité : O(1)
 * @param l liste */
void liberer_liste(Liste l);

/** @brief Ajoute un élément à la fin de la liste
 * Un équivalent de append en python.
 * Complexité **amortie** : O(1)
 * @param l liste,
 * @param x élément à ajouter à la fin.
 *
 *
 */
void ajouter_en_fin(Liste l, type_base x);

/**
 * @brief Renvoie l'élement d'une liste à une position donnée.
 * Complexité : O(1)
 * @param l liste,
 * @param pos index de l'élement à renvoyer (nombre négatif possible)
 * @returns l'élement qui se situe à l'index 'pos'.
 */
type_base element(Liste l, int pos);

/**
 * @brief Modifie l'élement d'une liste à une position donnée.
 * Complexité : O(1)
 * @param l liste,
 * @param pos index de l'élément à modifier.
 * @param nouvelle_valeur ce par quoi on veut remplacer la valeur.
 */
void modifier(Liste l, int pos, type_base nouvelle_valeur);

/**
 * @brief Échange deux éléments d'une liste étant données leurs positions.
 * **Complexité :** O(1)
 * @param l liste,
 * @param pos1 index du premier élément dont on souhaite changer la position,
 * @param pos2 index du second élément dont on souhaite changer la position.
 */
void echanger(Liste l, int pos1, int pos2);
```