CONTRÔLE TERMINAL 2023 SECONDE SESSION
(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 (5 points)

Q1. Rappelez les différentes complexités amorties asymptotiques pour un tas binaire minimum :

- Ajouter un élément ;
- Renvoyer le minimum ;
- Supprimer le minimum.

Q2. On définit les compteurs binaires comme des listes chaînées de `0` et de `1` représentant des écritures binaires de nombres (à l'envers).

Quelle est la complexité amortie de la fonction ci-dessous ?

```
Incremente(L:Compteur Binaire)
        Si L == NULL
        Alors   Renvoyer Noeud avec pour valeur 1
        Sinon   Si L->valeur == 0
                Alors   L->valeur = 1
                Sinon   L->valeur = 0
                        L->suivant = Incremente(L->suivant)
                Renvoyer L
```

Pour s'aider, on pourra utiliser la fonction de potentiel

$$Potentiel(L) = \text{nombre de chiffres 1 dans } L.$$

Questions pratiques (11 pts)

Voici ce que le générateur de sujet m'a pondu :

```
>>> genere_sujet()
[('tabdyn', 'liberer_liste', 4), ('unionfind', 'unir', 20), ('avl', 'inserer_avl
-insertion_noeud', 15), ('ensemble', 'lc+ensemble_vers_liste', 10)]
```

EXERCICE 1 – Libération d'un tableau dynamique (1 pt)

Écrire la fonction `liberer_liste` dont le prototype se décrit comme ci-dessous :

```
/**
 * @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);
```

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

```
struct TableauDynamique {

        type_base* tableau; /**< L'adresse du bloc mémoire
                                où se situent les valeurs.*/

        size_t taille;   /**< Le nombre actuel d'éléments. */

        size_t capacite; /**< Le nombre maximum d'éléments que peut stocker
        la structure sans faire de réallocation. */

        size_t (*fonction_calcul_capacite) (size_t); /**< Un pointeur vers la
        fonction qui indique comment la nouvelle capacité à partir de l'ancienne. */

};

typedef struct TableauDynamique* Liste;
```

EXERCICE 2 – Ensemble vers liste (3 pts)

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

```
/**
 * @brief Convertit une liste chaînée en une liste.
 * La liste chaînée n'est pas modifiée.
 * **Complexité :** O(taille de la liste chaînée)
 * @param l une liste chaînée.
 * @returns une liste qui contient les mêmes éléments que l
 */
Liste liste_chainee_vers_liste(ListeChainee l);
```

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;
```

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.

Q2. Écrire la fonction `liste_chainee_vers_liste` avec pour prototype :

```
 * @brief Convertit un ensemble en une liste.
 * L'ensemble n'est pas modifié.
 * **Complexité :** O(taille de l'ensemble)
 * @param e un ensemble.
 * @returns une liste contenant les mêmes éléments que **e** (l'ordre n'a
 * pas d'importance).
 */
Liste ensemble_vers_liste(Ensemble e);
```

Pour manipuler les listes chaînées dans cette fonction, on utilisera seulement la fonction précédente (`liste_chainee_vers_liste`).

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;
```

EXERCICE 3 – Union (3.5 pts)

Écrire la fonction `unir` avec pour prototype :

```
/**
 * @brief Fusionne deux ensembles étant donnés
 * deux éléments dans chacun des ensembles.
 * On implémentera l'union où on greffera l'arbre
 * le plus petit sur l'arbre le plus gros.
 * On provoquera une erreur si jamais un des deux
 * éléments n'est pas dans la partition.
 * On ne fera rien si les éléments appartiennent
 * dans le même ensemble.
 * **Complexité (amortie) :** alpha(n),
 * où alpha est l'inverse de la fonction Ackermann,
 * et n le nombre d'éléments dans la partition.
 * @param p une partition,
 * @param x un élément du premier ensemble,
 * @param y un élément du second ensemble.
 */
void unir(Partition p, int x, int y);
```

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. On utilisera la fonction `trouver` qu'on supposera déjà écrite. Son prototype est :

```
/**
 * @brief Renvoie un représentant d'un ensemble, étant donné un élément
 * dans cet ensemble.
 * Ici, le représentant correspond à la racine de l'arbre où se trouve l'élément.
 * Implémente l'opération "find" avec compression de chemins.
 * On provoquera une erreur si jamais l'élément n'est pas dans la partition.
 * **Complexité (amortie) :** alpha(n),
 * où alpha est l'inverse de la fonction Ackermann,
 * et n le nombre d'éléments dans la partition.
 * @param p une partition,
 * @param x un élément.
 * @returns la racine de l'arbre où se trouve x dans p.
 */
int trouver(Partition p, int x);
```

EXERCICE 4 – Insertion d'un noeud (3.5pts)

Écrire la fonction `insertion_noeud_premiere_etape` avec pour prototype :

```
/**
 * @brief Première étape de l'insertion d'une feuille dans un avl.
 * On insèrera une feuille sans mettre à jour les facteurs d'équilibrage.
 * On renverra un pointeur sur le dernier noeud
 * de facteur d'équilibrage non nul (le noeud le plus bas)
 * **Complexité :** O(log(nombre de noeuds))
 * @param a un avl,
 * @param x la valeur du noeud qu'on veut insérer.
 * @returns l'adresse du noeud le plus bas
 * qui a un facteur d'équilibrage différent de 0.
 */
avl insertion_noeud_premiere_etape(avl a, int x);
```

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 – Îles flottantes

Vous habitez dans une archipel avec n îles, sur l'île joliment appelée *Île 1*. Les autres îles s'appellent (vous l'avez deviné) *Île 2*, *Île 3*...

À l'origine, toutes les îles étaient isolées entre elles. On a rajouté des ponts au fur et à mesure des années afin de les interconnecter.

On donne une liste de couples d'entiers `ponts` de sorte que `ponts[i]` a été chronologiquement le `(i+1)`-ième pont à être installé sur l'archipel. Si `ponts[i]` est le couple (j, k), alors cela veut dire que ce `(i+1)`-ième pont relie *Île* j et *Île* k.

Le problème est de déterminer combien de ponts devront être construits avec que vous puissez rejoindre *Île 2* depuis *Île 1*.

Par exemple, si la liste de ponts est

```
[(4,6),(3,5),(4,2),(2,6),(1,7),(1,6),(6,7),(1,2)]
```

alors *Île 1* et *Île 2* ont été reliées du moment que le sixième pont, à savoir $(1, 6)$, a été installé (on emprunte ensuite $(6, 2)$ pour rejoindre *Île 2*). Donc la bonne réponse est `6`.

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);
```