Révisions - Programmation C

I. Pied à l'étrier

1. Écrire une fonction `void swap(int* a, int* b)` qui prend en entrée deux pointeurs vers des entiers et échange les valeurs pointées.

2. Écrire une fonction `int pgcd(int a, int b)` qui calcule le pgcd entre deux entiers à l'aide de l'algorithme d'Euclide. On attend une implémentation itérative.

3. Écrire une fonction `bool prefix(char* c1, char* c2)` déterminant si une chaîne de caractères `c2` est préfixe de `c2`.

II. Manipulation de tableaux

4. Écrire une fonction `int* creer_tableau(int n, int x)` qui crée et renvoie un tableau de taille `n` contenant la valeur `x` dans toutes ses cases.

5. Écrire une fonction `int* tirage(int n, int a, int b)` retournant un tableau de n entiers aléatoires tirés uniformément dans $[\![a, b]\!]$.

6. Écrire une fonction `int indice(int* tab, int n)` calculant l'indice du premier `0` dans un tableau d'entiers parcouru par une boucle `while` arrêtée dès que possible, sans utiliser de `break`, ni de `return` dans la boucle.
La fonction devra retourner `-1` si le tableau ne contient aucun `0`.

7. Écrire une fonction `int majoritaire(int* tab, int n, int m)` tel que `majoritaire(tab, n, m)`, renvoie l'entier de $[\![0, m-1]\!]$ ayant le plus d'occurrences dans le tableau de taille n. On fournira une implémentation en $O(n+m)$.

III. Manipulation de types personnalisés

Listes chaînées

On définit les types suivants :

```
struct maillon {
    struct maillon* next;
    char* elem;
};

typedef struct maillon* liste;
```

8. Écrire une fonction `liste ajout_tete(liste lst, char* s)` permettant d'ajouter un élément en tête dans une liste chaînée, et renvoyant la nouvelle liste.

9. Écrire une fonction `liste ajout_queue(liste lst, char* s)` permettant d'ajouter un élément à la toute fin dans une liste chaînée, et renvoyant la nouvelle liste.
Écrire une version récursive et une version itérative.

10. Écrire une fonction `void print_list(liste lst)` affichant les éléments d'une liste chaînée.
Écrire une version récursive et une version itérative.

11. Écrire une fonction `int indice (liste lst, char* s)` qui renvoie la position de l'élément s dans une liste chaînée. Si l'élément n'est pas présent, on renverra -1. On utilisera `strcmp` pour comparer les chaînes.
Écrire une version récursive et une version itérative.

12. Écrire une fonction `liste retire(liste lst, char* s)` permettant de retirer un élément présent dans la liste.

13. Écrire une fonction `liste renverse(liste lst)` qui renvoie une nouvelle liste correspondant à la liste `lst` dont l'ordre des éléments a été inversé.
Écrire une version récursive et une version itérative.

14. Écrire une fonction `void free_list(liste lst)` permettant de supprimer du tas tous les maillons de la liste.

Tables de hachage

Dans cette partie, on implémente une table de hachage permettant de stocker des chaînes de caractères.
On suppose disposer d'une fonction `int hach(char* s)`, et on gère les collisions par chaînage, à l'aide des types `struct maillon` et `liste` précédemment définis.
On suppose finalement que `N` est une constante littérale valant 100.

```
#define N 100

typedef struct table_h {
    liste tab [N];
} table_h;
```

15. Écrire une fonction `table_h* init()` qui renvoie l'adresse d'une nouvelle table de hachage.

16. Écrire une fonction `void ajoute(table_h* th, char* s)` qui ajoute un élément dans un table de hachage.
On rappelle que les collisions sont gérées par chaînage.

17. Écrire une fonction `bool est_present(table_h* th, char* s)` qui indique si un élément est présent dans un table de hachage.

18. Écrire une fonction `void retire(table_h* th, char* s)` qui enlève un élément présent dans un table de hachage.

Tas min

Pour implémenter un tas min, on utilise la représentation par tableau issue du parcours en largeur. On définit le type suivant qui permet de stocker des entiers :

```
typedef struct tas {
    int* tab;
    int capacite;
    int taille;
} tas;
```

19. Écrire une fonction `tas* init(int capacite)` qui crée un tas vide, de capacité donnée en argument.

20. Écrire une fonction `void ajout(tas* t, int x)` qui ajoute un élément dans un tas. On s'assurera à l'aide d'une assertion que le tas n'est pas plein, et on veillera à ce que la propriété de tas soit respectée.

21. Écrire une fonction `int minimum (tas* t)` qui renvoie et supprime la valeur du minimum du tas.

22. Écrire une fonction `void tri_tas (int* tab, int n)` qui prend en entrée un tableau d'entiers et le trie en utilisant l'algorithme du tri par tas.

IV. Manipulation de graphes

Matrice d'adjacence

Pour représenter un graphe orienté par matrice d'adjacence, on propose le type suivant :

```
typedef struct graphe_mat {
    int n; //nbr de sommets
    bool** tab; //tab[i][j] indique si l'arc (i,j) existe
} graphe_mat;
```

23. Écrire une fonction `graphe_mat* cree_graphe_vide(int n)` qui renvoie un pointeur vers un nouveau graphe à n sommets et sans arcs.

24. Écrire une fonction `void supprime(graphe_mat* g)` qui prend en argument l'adresse d'une structure de graphe et libère l'espace mémoire qu'elle occupe.

25. Écrire une fonction `void ajoute_arc(graphe_mat* g, int i, int j)` qui ajoute, si besoin est, l'arc reliant ces deux sommets au graphe.

26. Écrire une fonction `int degre_sortant(graphe_mat* g, int s)` qui renvoie le degré sortant d'un sommet.

27. Écrire une fonction `int degre_entrant(graphe_mat* g, int s)` qui renvoie le degré entrant d'un sommet.

Algorithme de Floyd-Warshall

28. Modifier le type `struct graphe_mat` pour représenter des graphes orientés et pondérés.

29. À l'aide de ce type modifé, écrire une fonction `graphe_mat* floyd_warshall(graphe_mat* g)` qui prend en entrée un graphe orienté et pondéré et renvoie un matrice représentant les plus courts chemins entre toutes paires de sommets.

30. Écrire une fonction `bool contient_cycle_absorbant (graphe_mat* g)` qui prend en entrée un graphe orienté et pondéré et renvoie un booléen indiquant si le graphe admet un cycle de poids négatif.

Liste d'adjacence

Pour représenter un graphe par liste d'adjacence, on utilise la structure `graphe_lst` suivante :

```
typedef struct graphe_lst {
    int n;
    int deg_max;
    int** voisins;
} graphe_lst;
```

Si `g` est un objet de ce type, alors il représente un graphe à `g.n` sommets, dont le degré sortant maximal majoré par `g.deg_max`. Le tableau `g.voisins` est de dimensions `n` $\times$ `deg_max`, et `g.voisins[i][k]` stocke le k-ème voisin sortant du sommet i, et -1 si un tel voisin n'existe pas.

31. Écrire une fonction `graphe_lst* cree_graphe_vide(int n, int deg_max)` qui renvoie un pointeur vers un nouveau graphe à n sommets et sans arcs.

32. Écrire une fonction `void supprime(graphe_lst* g)` qui prend en argument l'adresse d'une structure de graphe et libère l'espace mémoire qu'elle occupe.

33. Écrire une fonction `void ajoute_arc(graphe_lst* g, int i, int j)` qui ajoute, si besoin est, l'arc reliant ces deux sommets au graphe. Si l'ajout est impossible, la fonction ne fera rien.

34. Écrire une fonction `int degre_sortant(graphe_lst* g, int s)` qui renvoie le degré sortant d'un sommet.

35. Écrire une fonction `int degre_entrant(graphe_lst* g, int s)` qui renvoie le degré entrant d'un sommet.