

## Contrôle exemple d'algorithmique (Correction à la fin)

Ce sujet est donné à but d'entraînement, le contrôle comportera plus de questions et d'exercices.

Pour toute question ou remarque relative aux cours : pierre.bertin-johannet@orange.fr

### Exercice 0: QCM

Plusieurs réponses possibles, pas de points négatifs.

**1. Une liste doublement chaînée est:**

- a) Un type particulier de table de hachage
- b) Une liste de caractères
- c) Une structure contenant deux listes
- d) Une liste dans laquelle chaque élément contient deux liens vers le suivant

**2. Un arbre binaire de recherche est :**

- a) Un arbre dans lequel la valeur des noeuds est inférieure à celle du noeud parent
- b) Un arbre dans lequel la valeur du noeud parent est comprise entre celle du noeud gauche et celle du noeud droit
- c) Un arbre dans lequel chaque noeud a deux fils
- d) Un arbre complet

**3. Si on insère les élément 3, 2 puis 1 dans une file f :**

- a) défiler(f) renverra l'élément 3
- b) Enfiler un élément ne changera pas le résultat du prochain appel à la fonction défiler
- c) Je peux défiler trois fois sans causer d'erreurs dans mon programme

**4. Le nombre d'opérations requises pour accéder à l'élément en position k d'une liste simplement chaînée :**

- a) Varie selon la taille de la liste
- b) Est constant
- c) Varie selon k

**5. Le nombre d'opérations requises pour retirer l'élément en position k d'une liste chaînée de taille n :**

- a) Varie selon la taille de la liste
- b) Est constant
- c) Varie selon k

**6. Le nombre d'opérations requises pour accéder au dernier élément d'un tableau de taille n :**

- a) Varie selon la taille du tableau
- b) Est constant
- c) Varie selon n

**7. Le nombre d'opérations requises pour insérer un élément dans un tableau de taille n :**

- a) Varie selon le nombre d'éléments
- b) Est constant
- c) Varie selon n

**8. Le nombre d'opérations requises pour rechercher un élément dans un arbre binaire de recherche contenant n noeuds :**

- a) Est constant en moyenne
- b) Augmente proportionnellement au logarithme du nombre d'éléments ajoutés
- c) Augmente proportionnellement au nombre d'éléments ajoutés
- d) Augmente proportionnellement au carré du nombre d'éléments ajoutés

9. Un arbre binaire complet contenant 57 noeuds sera de hauteur :

- a) 5
- b) 6
- c) 7
- d) 8

### Exercice 1:

Le code suivant utilise une pile p dans laquelle ont été empilés les nombres 1, 2, 2, 6, 6, 3, 1 dans cet ordre (en commençant par empiler le nombre 1 et en terminant par empiler le nombre 1). **TODO** 1 et 1 really ?

```
int a = 0;
while(a != 6){
    a = depiler(f);
    if (a == 6){
        empiler(f, 2);
    }
    printf("%d\n", a);
}
```

- a) Qu'affichera le programme dans la console à l'exécution ?
- b) Que reste-t-il dans la pile après exécution du programme ?

### Exercice 2:

Soit le code suivant :

```
arbre* g = creer_arbre('+');
g->fils_gauche = creer_arbre('4');
g->fils_droit = creer_arbre('6');
arbre* d = creer_arbre('-');
d->fils_gauche = creer_arbre('7');
d->fils_droit = creer_arbre('1');
arbre* a = creer_arbre('*');
a->fils_gauche = g;
a->fils_droit = d;
```

- a) Dessinez un schéma de l'arbre produit par ce code.
- b) Qu'afficherait dans la console un programme qui effectuerait un parcours en profondeur infixe de l'arbre pour afficher la valeur des noeuds ?
- c) Ecrivez le code d'un parcours en largeur qui affichera les noeuds de cet arbre.

### Exercice 3:

Soit le programme suivant qui crée une liste de taille variable et y ajoute six éléments.

```
// Crée une nouvelle liste chaînée
// Fonction identique a l'implémentation faite en TP
LinkedList* linked_list_new(){
    LinkedList* l = (LinkedList*)malloc(sizeof(LinkedList));
    l->sentinel_start.next = &(l->sentinel_end);
    l->sentinel_end.next = &(l->sentinel_end); // la sentinelle de fin pointe vers elle même
    l->sentinel_start.prev = &(l->sentinel_start);
    l->sentinel_end.prev = &(l->sentinel_start); // la sentinelle de fin pointe vers elle même
    return l;
}

// insère un element contenant valeur après prev
// Fonction identique a l'implémentation faite en TP
```

```

void insere_apres_elem(ListElem* prev, int valeur){
    ListElem* new_elem = malloc(sizeof(ListElem));
    new_elem->valeur = valeur;
    new_elem->next = prev->next;
    new_elem->next->prev = new_elem;
    prev->next = new_elem;
    new_elem->prev = prev;
}

// On insère un élément dans une liste déjà triée
// Pour cela on cherche d'abord un nombre plus petit que le nombre à ajouter
// Ensuite on s'insère juste avant ce nombre
void insere_dans_liste_triee(LinkedList* list, int nombre){
    list->sentinel_end.valeur = nombre - 1;

    ListElem* elem = list->sentinel_start.next;
    while (elem->next != elem && elem->valeur < nombre){
        elem = elem->next;
    }
    insere_apres_elem(elem->prev, nombre);
}

int main(){
    LinkedList* list = linked_list_new();

    insere_dans_liste_triee(list, 1);
    insere_dans_liste_triee(list, 2);
    insere_dans_liste_triee(list, 3);
}

```

Lisez attentivement le code avant de répondre aux questions

### Partie I: Comptage

- Combien de fois la fonction malloc sera-t-elle appelée par ce programme ?
- Combien de fois la ligne “elem = elem->next;” sera-t-elle exécutée par ce programme ?
- Que deviennent les réponses aux deux questions précédentes si l’on ajoute les lignes suivantes à la fin de la fonction main ?

```

insere_dans_liste_triee(list, 4);
insere_dans_liste_triee(list, 5);
insere_dans_liste_triee(list, 6);

```

### Partie II: Complexité

- Comment évolue le nombre d’appels à la fonction malloc effectués par le programme donné ? Choisissez la réponse ci-dessous qui vous semble la plus proche.
  - Il est constant
  - Il augmente proportionnellement au nombre d’éléments ajoutés
  - Il augmente proportionnellement au carré du nombre d’éléments ajoutés
- Même question pour le nombre de fois que la ligne “elem = elem->next;” sera exécutée par le programme.

### Partie III: Arbre binaire de recherche

- Si on utilise un arbre binaire de recherche à la place de la liste chaînée, comment évoluera le nombre d’opérations ?

## Correction

### Exercice 0: QCM

Plusieurs réponses possibles, pas de points négatifs.

**1. Une liste doublement chaînée est:**

- a) Un type particulier de table de hashage
- b) Une liste de caractères
- c) Une structure contenant deux listes
- d) Une liste dans laquelle chaque élément contient deux liens vers le suivant

**2. Un arbre binaire de recherche est :**

- a) un arbre dans lequel la valeur des noeuds est inférieure à celle du noeud parent
- b) **un arbre dans lequel la valeur du noeud parent est comprise entre celle du noeud gauche et celle du noeud droit**
- c) un arbre dans lequel chaque noeud a deux fils
- d) **un arbre complet**

**3. Si on insère les élément 3, 2 puis 1 dans une file f :**

- a) a) **Défiler(p) renverra l'élément 3**
- b) b) **Enfiler un élément ne changera pas le résultat du prochain appel à la fonction défiler**
- c) c) **Je peux défiler trois fois sans causer d'erreurs dans mon programme.**

**4. Le nombre d'opérations requises pour accéder à l'élément en position k d'une liste simplement chaînée :**

- a) Varie selon la taille de la liste
- b) Est constant
- c) **Varie selon k**

**5. Le nombre d'opérations requises pour retirer l'élément en position k d'une liste chaînée de taille n :**

- a) Varie selon la taille de la liste
- b) Est constant
- c) **Varie selon k**

**6. Le nombre d'opérations requises pour accéder au dernier élément d'un tableau de taille n :**

- a) Varie selon la taille du tableau
- b) **Est constant**
- c) Varie selon n

**7. Le nombre d'opérations requises pour insérer dans un tableau de taille n :**

- a) **Varie selon le nombre d'éléments**
- b) Est constant
- c) **Varie selon n**

**8. Le nombre d'opérations requises pour rechercher un élément dans un arbre binaire de recherche contenant n noeuds :**

- a) Est constant en moyenne
- b) **augmente proportionnellement au logarithme du nombre d'éléments ajoutés**
- c) augmente proportionnellement au nombre d'éléments ajoutés
- d) augmente proportionnellement au carré du nombre d'éléments ajoutés

**9. Un arbre binaire complet contenant 57 noeuds sera de hauteur :**

- a) 5
- b) 6
- c) 7

d) 8

### Exercice 1:

```
int a = 0;
while(a != 6){
    a = depiler(p);
    if (a == 6){
        empiler(p, 2);
    }
    printf("%d\n", a);
}
```

a) Le programme affichera :

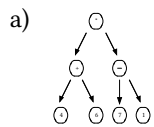
1

3

6

a) Il reste dans la file: 1, 2, 2, 6, 2

### Exercice 2:



b) Un parcours infixé en profondeur affichera : 4+6\*7-1

```
c) void parcours_largeur(Noeud* n){
    File* f;
    enfiler(f, n);
    while(taille(f)){
        Noeud* suivant = defiler(f);
        if (suivant != 0){
            enfiler(suivant->noeud_gauche);
            enfiler(suivant->noeud_droit);
            printf("%c\n", suivant->valeur);
        }
    }
}
```

### Exercice 3:

#### Partie I Comptage

a) La fonction malloc est appelée quatre fois par ce programme

b) La ligne "elem = elem->next;" est appelée :

- Zero fois pour insérer le premier élément car on l'insère au début
- Une fois pour insérer le second élément car on l'insère en deuxième position
- Deux fois pour insérer le troisième élément car on l'insère en troisième position

Elle est donc appelée 3 fois.

c) Si on insère 4, 5 et 6 on appellera la fonction malloc trois fois de plus. Quand à la ligne "elem = elem->next" elle sera appelée :

- Trois fois pour insérer le 4
- Quatre fois pour insérer le 5
- Cinq fois pour insérer le 6

Elle aura donc au total été appelée :  $1+2+3+4+5 = 15$  fois.

## Partie II Complexité

- On ne fait qu'un seul malloc pour chaque élément ajouté donc :
  - Il est constant
  - **Il augmente proportionnellement au nombre d'éléments ajoutés**
  - Il augmente proportionnellement au carré du nombre d'éléments ajoutés
- On calcule le nombre de fois que sera appelée la ligne "elem = elem->next;" pour insérer **n** éléments.

Pour chaque élément il faut exécuter la ligne assez de fois pour trouver dans la liste un élément plus petit que celui qu'on cherche à insérer.

Dans le pire des cas on parcourt toute la liste pour s'insérer à la fin.

Le parcours de la liste demande **0** opérations pour le premier élément et **n - 1** opérations pour le dernier, il est donc en moyenne **(n-1)/2**

Étant donné qu'il faut effectuer cette opération **n** fois, le nombre d'opérations pour insérer **n** éléments sera **n\*((n-1)/2)** soit  $\frac{n^2-n}{2}$

Donc :

- Il est constant
- Il augmente proportionnellement au nombre d'éléments ajoutés
- **Il augmente proportionnellement au carré du nombre d'éléments ajoutés**

## Partie III Arbre binaire de recherche

- L'insertion d'un élément dans un arbre binaire de recherche contenant **n** noeuds est proportionnelle au logarithme du nombre de noeuds.

Étant donné qu'il faut effectuer cette opération **n** fois, le nombre d'opérations est proportionnel à  $n * \log_2(n)$