Examen : Structures de données

Vayssade Jehan-Antoine

26 mars 2026

Durée : 1 heures 30 Note maximal : 20 points
Matériel autorisé : aucun document, calculatrice interdite.
Consignes : Justifiez toutes vos réponses. Les codes doivent être écrits en C.

1 Complexités et recherche (7.0 points - 30 min)

- (1.5 pts) Expliquez les intérêts et les limites de l'analyse asymptotique et définissez les trois notations asymptomatiques et leurs rôles respectifs.
- (3.0 pts) Implémentez en C les fonctions suivantes.
  - `int dichotomic_search_recursive(int *arr, int n, int key);`
  - `int dichotomic_search_iterative(int *arr, int n, int key);`
- (0.5 pts) Que permet de faire une recherche par interpolation ?
- (2.0 pts) Quels sont les coûts additionnels des fonctions récursives et leurs impacts sur les complexités temporelle et spatiale ? Évaluez en considérant une profondeur de récursion d. Comparez la complexité sur vos implementations précédentes.

2 Construction et dégénérescence (4.5 points - 15 min)

On considère la séquence d'insertions suivante dans un BST initialement vide :

$$50, 30, 70, 20, 40, 60, 80, 10$$

1. (0.5 pt) Dessinez l'arbre binaire de recherche (en respectant l'ordre donné).
2. (0.5 pt) Cet arbre est-il équilibré et/ou complet ?
3. (0.5 pt) Donnez une permutation qui conduirait à un arbre **dégénéré**.
4. (0.5 pt) Quelle est la propriété fondamentale qu'un BST doit satisfaire ?
5. (0.5 pt) Dans un BST contenant n nœuds distincts, quelle est la hauteur minimale et maximale possible ? Exprimez vos réponses en fonction de n.
6. (0.5 pt) Citez une structures de données équilibrées qui garantissent une bonne complexité pour les opérations de base.
7. (1.5 pts) Proposez une stratégie simple (sans modifier la structure de base du BST) qui permet de réduire fortement la probabilité d'obtenir un arbre dégénéré. En considèrent l'insertion de toutes les valeurs d'un tableau. Expliquez pourquoi cette méthode fonctionne. Vous pourrez partir d'une situation de pire cas pour démontrer l'intérêt. Finalement déduisez la complexiez temporel de votre solution.

3 Vérification et parcours (4.0 points - 20 min)

On donne la structure suivante :

```
typedef struct BinaryTreeNode {
    int key;
    struct BinaryTreeNode *left;
    struct BinaryTreeNode *right;
} BinaryTreeNode;
```

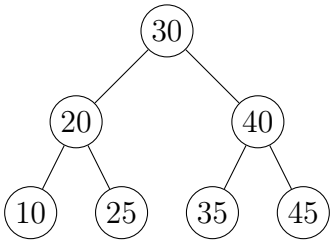
1. (1.25 pts) Écrivez une fonction **récursive** qui vérifie si un arbre passé en paramètre est un BST valide. La fonction doit retourner `1` si l'arbre est un BST correct, `0` sinon.

```
int is_banary_search_tree(BinaryTreeNode *racine);
```

2. (2.0 pts) Écrivez une fonction **récursive** qui vérifie si un arbre passé en paramètre est équilibré. La fonction doit retourner `0` si l'arbre est équilibré, et sinon retourner h, c'est-à-dire la différence de hauteur entre les sous-arbres.

```
int is_balanced(BinaryTreeNode *root)
```

3. (0.75 pts) Donnez trois parcours (préfixe, infixe, postfixe) sur l'arbre suivant :



4 Heapsort et tas binaires (4.5 points - 15 min)

On rappelle que Heapsort utilise un tas **binaire maximal** (max-heap) pour trier un tableau en place.

1. (1.0 pt) Soit le tableau initial suivant : $[15, 8, 12, 20, 3, 7, 18, 1, 10]$. Montrez l'état du tableau après la phase **build-heap** (tas maximal). Vous pouvez représenter le tableau sous forme d'arbre ou ligne par ligne.
2. (2.5 pts) Écrivez la fonction `void heapsort(int A[], int n)` complète qui trie le tableau `A[0..n[` en ordre croissant en utilisant un max-heap. Implémentez la phase d'extraction successive du maximum (n-1 fois). Vous pouvez utiliser les fonctions ci-dessous comme primitives déjà implémentées :
   — `void percolate_down(int A[], int n, int i);`
   — `void build_heap(int A[], int n);`
3. (1.0 pt) Quelle est la complexité temporelle de Heapsort dans le meilleur, moyen et pire cas ? Epliquez la complexité de la construction du tas et de la phase d'extraction.

5 B-arbres (5.0 points - 20 min)

1. (1.0 pts) Dessinez l'arbre après chaque insertion qui provoque une scission (split). Indiquez clairement les nœuds qui éclatent. A partir des clés suivantes, dans un arbre initialement vide :

$$10, 20, 5, 15, 30, 25, 7, 12, 27, 32, 3$$

2. (1.0 pt) Un B-arbre d'ordre $m = 101$ (fréquent pour les index de bases de données) contient environ 1 million de clés. Estimez la hauteur maximale de l'arbre (en nombre de niveaux). Comparez avec un arbre binaire équilibré. Pour faciliter les calcules vous pouvez faire des approximations sur les bornes min-max (eg : 101->100 ; 49-51 -> 50).
3. (0.5 pt) Expliquez pourquoi les opérations d'insertion et de suppression dans un B-arbre nécessitent parfois des **fusions** ou des **emprunts** (borrow) entre nœuds frères.
4. (2.5 pt) Montrez que la hauteur h d'un B-arbre contenant n clés vérifie :

$$h \leq \log_{\lceil m/2 \rceil} \left(\frac{n+1}{2} \right) + 1$$

6 Analyse et débogage (5.0 points - 20 min)

On considère l'implémentation itérative suivante de la Tour de Hanoï :

1. (1.5 pt) Quelle est la complexité temporelle asymptotique de cet algorithme ?
2. (1.5 pt) Quelle est la complexité spatiale (taille maximale de la pile) dans le pire cas ?
3. (2.0 pts) Ce code contient plusieurs erreurs graves liées à la gestion dynamique de la mémoire. Identifiez et expliquez au moins **quatre (4) problèmes distincts**.

```c
typedef struct { int n, phase; char src, aux, dst;} Task;

void hanoi_iteratif(int n, char src, char aux, char dst) {
    int capacity = 16, top = -1;
    Task *stack = malloc(capacity * sizeof(Task));
    stack[++top] = (Task){n, 0, src, aux, dst};

    while (top >= 0) {
        Task *t = &stack[top];
        if (t->n == 1) { top--; continue;}
        if (t->phase == 0) {
            t->phase = 1;
            if (top + 1 >= capacity) {
                capacity *= 2;
                stack = realloc(stack, capacity * sizeof(Task));
            }
            stack[++top] = (Task){t->n-1, 0, t->src, t->dst, t->aux};
        }
        else if (t->phase == 1) {
            t->phase = 2;
            if (top + 1 >= capacity) {
                capacity *= 2;
                stack = realloc(stack, capacity * sizeof(Task));
            }
            stack[++top] = (Task){t->n-1, 0, t->aux, t->src, t->dst};
        }
        else top--;
    }
}
```