Correction : Examen Structures de données

Vayssade Jehan-Antoine

26 mars 2026

Durée : 1 heure 30 Note maximale : 20 points
Consignes : Justifiez toutes vos réponses. Les codes doivent être écrits en C.

1 Complexités et recherche (7,0 points, jusqu'à 11)

(1,5 pts) L'analyse asymptotique permet d'étudier le comportement d'un algorithme lorsque la taille des données $n \to \infty$. **Intérêts :** indépendance de la machine, du compilateur, des optimisations et des cas particuliers ; comparaison objective d'algorithmes. **Limites :** ne rend pas compte des constantes cachées, des cas particuliers (meilleur/-pire), ni des comportements pour de petites valeurs de n.

Les trois notations :

— $O(f(n))$: majoration asymptotique (borne supérieure) ;
— $\Omega(f(n))$: minoration asymptotique (borne inférieure) ;
— $\Theta(f(n))$: encadrement asymptotique (borne serrée).

En pratique cela peu permettre de choisir quel algorithme choisir en fonction des contraintes (soft). Faut-il optimiser l'utilisation mémoire ? ou le temps d'exécution ? Mais ne permet de de résoudre la question des contraintes (hard). Tel-que nombre de cycle cpu exacte, mémoire exacte etc. Pour cela il existe de meilleurs outils mathématiques tel-que la sémantique axiomatique. Mais aussi, plus simplement des benchmarks (Analyse expérimentale). Qui permettent de ce rendre compte de comme l'algorithme ce comporte sur de petites tailles, ce qui nous intéresse dans la majorités des cas. Ou encore l'analyse probabiliste qui testes plusieurs distributions de donnes pour une même tailles afin d'extraire les moyennes et écarts-types (+1.5 pts)

(3,0 pts) Implémentation de la recherche dichotomique :

```c
int dichotomic_search_recursive(int *arr, int n, int key) {
    if (n <= 0) return -1;
    int mid = n / 2;
    if (arr[mid] == key) return mid;
    if (arr[mid] > key)
        return dichotomic_search_recursive(arr, mid, key);
    else
        return dichotomic_search_recursive(arr + mid + 1, n - mid -
   1, key);
}

int dichotomic_search_iterative(int *arr, int n, int key) {
    int low = 0, high = n - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == key) return mid;
        if (arr[mid] > key)
            high = mid - 1;
        else
            low = mid + 1;
    }
    return -1;
}
```

(0,5 pt) Une recherche par interpolation permet d'estimer la position probable de la clé en utilisant une interpolation linéaire (comme dans la recherche dans un annuaire). Elle est particulièrement efficace lorsque les valeurs sont uniformément distribuées. Le nombre d'itération dans ce cas de figure est typiquement autour de 2 voir 3. Largement meilleur que la recherche dichotomique, a condition d'avoir les bonne propriétés mathématiques sur l'ensemble.

Dans le pire cas, si les donnes sont très très mal distribues on ce rapprochera de $O(n)$, exemple ([1, 2, 3, 4, 5, 6, 7, 8, 1000000]). Dans le cas de donnes juste mal distribuées on pourra supposer une complexités en $log_t(n)$ et dans le cas moyen en $log(log(n))$. Finalement le meilleur cas est en $O(1)$ lorsque les données sont linéairement distribuées. (+1 pts)

(2.0 pts) Le coûts additionnels des fonctions récursives : **Temps :** chaque appel ajoute le coût de la gestion de la pile (sauvegarde des registres, passage des paramètres, adresse de retour). **Espace :** $O(d)$ où d est la profondeur de récursion (pile d'appels).

Pour la recherche dichotomique :

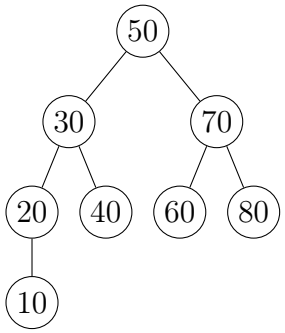
— Version itérative : temps $O(\log n)$, espace $O(1)$.

— Version récursive : temps $O(d * \delta)$ (avec d nombre d'appels), espace $O(d * \delta)$ à cause de la profondeur $d \approx \log_2 n$. Le terme δ est ajouté pour modéliser les coûts additionnels invisibilisé (temps d'appelle de la fonction, espace mémoire pour les copies locales etc). (+0.5 pt avec δ) Si le coût d'appel de la fonction est 20 cycles alors $\delta = 20$ dans l'estimation de la complexité temporel. Réciproquement en mémoire pour représenter les coûts additionnels de stockages. Le terme δ est une constante et finalement supprimé dans l'analyse de la complexité asymptomatique, résultant en une perte d'information. Et donc une sous-estimation des coûts réels, ce qui renforce les arguments de la question 1.1 (+1 pt)

2 Construction et dégénérescence (4,5 points , jusqu'à 6)

Séquence : 50, 30, 70, 20, 40, 60, 80, 10

1. Arbre :



2. L'arbre est équilibré, car la hauteur du sous-arbre gauche est de 3 et celle du sous-arbre droit est de 2. La différence étant inférieure ou égale à 1, la condition d'équilibre est respectée. C'est la même propriété qui est utilisée dans les arbres AVL pour déterminer si des rotations sont nécessaires ($h \in \{-1, 0, 1\}$) (+0.5 pt). L'arbre est également complet. Cela signifie que chaque niveau est entièrement rempli, sauf éventuellement le dernier, qui est rempli de gauche à droite. Autrement dit, l'arbre peut être stocké dans un tableau contenant le même nombre de cases que de nœuds, même principe qu'un tas binaire utilisé, ex dans heapsort (+0.5 pt).
3. Une permutation qui conduit à un arbre dégénéré (liste chaînée) : par exemple l'ordre croissant 10, 20, 30, 40, 50, 60, 70, 80. C'est une permutation total des éléments, pas juste un élément ...
4. Propriété fondamentale d'un BST : pour tout nœud, toutes les clés du sous-arbre gauche sont inférieures à la clé du nœud, et toutes celles du sous-arbre droit sont supérieures. Pas uniquement localement voir 3.1 (+0.5).
5. Dans un BST à n nœuds distincts :
   — Hauteur minimale : $\lfloor \log_2 n \rfloor + 1$ (arbre équilibré).
   — Hauteur maximale : n (arbre dégénéré).
6. Exemples de structures équilibrées : AVL, tas, etc.
7. Stratégie simple : mélanger aléatoirement le tableau avant d'insérer les éléments dans le BST (ou insérer dans un ordre aléatoire). **Pourquoi ça marche ?** Dans le pire cas (trié), on obtient une liste chaînée ($O(n)$ par opération). En permutant aléatoirement, la probabilité d'obtenir un arbre très déséquilibré devient très faible. En moyenne, la hauteur devient $O(\log n)$. Complexité temporelle attendue de la construction : $O(n \log n)$ .... Par contre cela ne permet tout de même pas d'obtenir un arbre équilibré ! On évite simplement les cas critiques. Pour faire mieux il faut regarder du coté de vrais algorithmes tel-que les arbres AVLs.

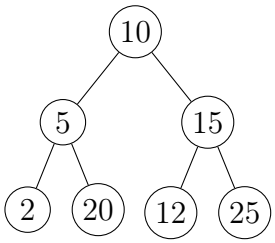
3 Vérification et parcours (4,0 points)

1. Fonction `is_binary_search_tree` (récursive avec bornes) :

```c
int is_bst_util(BinaryTreeNode *node, int min, int max) {
    if (node == NULL) return 1;
    if (node->key <= min || node->key >= max) return 0;
    return is_bst_util(node->left, min, node->key) &&
           is_bst_util(node->right, node->key, max);
}

int is_binary_search_tree(BinaryTreeNode *racine) {
    /* pour l'exercice nous pouvions aussi utiliser 0, 100
    a la place de INT_MIN, INT_MAX, en justifiant
    ou descendre tout l'arbre a gauche et a droite
    pour prendre respectivement les bornes min/max*/
    return is_bst_util(racine, INT_MIN, INT_MAX);
}
```

Attention vérifier uniquement, localement, si $left < key$ et $right > key$ n'est pas suffisant, car il est possible d'avoir un arbre qui accepte cette condition sans être un arbre de recherche binaire (BST). Par exemple, ici 20 pose soucis :



2. Fonction `is_balanced` (retourne 0 si équilibré, sinon la différence de hauteur) :

```c
int height(BinaryTreeNode *root) {
    if (root == NULL) return 0;
    int hl = height(root->left);
    int hr = height(root->right);
    return 1 + (hl > hr  ? hl : hr);
}
int is_balanced(BinaryTreeNode *root) {
    if (root == NULL) return 0;
    int left_h = height(root->left);
    int right_h = height(root->right);
    int diff = left_h - right_h;
    if (diff < 0) diff = -diff;
    if (diff > 1) return diff;
    int left_res = is_balanced(root->left);
    int right_res = is_balanced(root->right);
    if (left_res != 0) return left_res;
    if (right_res != 0) return right_res;
    return 0;
}
```

3. Parcours sur l'arbre donné :
   - Préfixe (pré-ordre) : 30, 20, 10, 25, 40, 35, 45
   - Infixe (in-ordre) : 10, 20, 25, 30, 35, 40, 45
   - Postfixe (post-ordre) : 10, 25, 20, 35, 45, 40, 30

4 Heapsort et tas binaires (4,5 points)

1. Tableau initial : [15, 8, 12, 20, 3, 7, 18, 1, 10] Après **build-heap** (max-heap) : [20, 15, 18, 10, 3, 7, 12, 1, 8]
2. Fonction `heapsort` :

```c
void heapsort(int A[], int n) {
    if (n <= 1) return;
    build_heap(A, n);           /* phase de construction */

    for (int i = n - 1; i > 0; i--) {
        /* echange racine avec le dernier element */
        int temp = A[0];
        A[0] = A[i];
        A[i] = temp;

        percolate_down(A, i, 0);   /* on reduit la taille du tas */
    }
}
```

3. Complexité de Heapsort :
   — Construction du tas (`build_heap`) : $O(n)$ (pas $O(n \log n)$ + 0.5 pt).
   — Phase d'extraction : $n - 1$ extractions, chacune $O(\log n) \rightarrow O(n \log n)$.
   — Total : $O(n \log n)$ dans le meilleur, moyen et pire cas (pas de cas meilleur particulièrement favorable).

5 B-arbres 5,0 points, jusqu'à 8

1. (Dessin des splits) : À réaliser sur feuille (insertions successives de 10, 20, 5, 15, 30, 25, 7, 12, 27, 32, 3 avec ordre $m = 5$). Les nœuds qui éclatent sont ceux qui dépassent $m - 1 = 4$ clés.
2. Pour $m = 101$ ($\lceil m/2 \rceil \approx 50$), environ 1 million de clés.

   Hauteur maximale $H \approx \log_{50}(10^6) \approx \frac{\log_{10}(10^6)}{\log_{10}(50)} \approx \frac{6}{\log_{10}(50)}$. Concernant $\log_{10}(50)$, nous savons que $\log_{10}(10) = 1$ et $\log_{10}(100) = 2$. Nous pouvons donc encadrer $\log_{10}(50)$ entre 1.5 et 1.75 à la louche pour les approximations de calculs. Nous avons donc $H \in \left[\frac{6}{1.75}, \frac{6}{1.5}\right] \approx [3.4, ; 4]$. L'arbre a donc entre 3 et 4 niveaux.

   Un arbre binaire équilibré aurait une hauteur $H \approx \log_2(10^6) = \frac{\log_{10}(10^6)}{\log_{10}(2)} = \frac{6}{\log_{10}(2)}$. Avec la même démarche nous savons que $\log_{10}(1) = 0$ ce qui permet de supposer que $\log_{10}(2) \approx 0.3$. On en déduis alors, que $H \approx \frac{6}{0.3} \approx 20$. L'arbre a donc entre 19 et 20 niveaux.

   (Démarche mathématique +1 pt)

   Ainsi, un arbre binaire équilibré contenant environ 10^6 clés aurait une hauteur d'environ 20 niveaux, tandis que le B-tree d'ordre 50 ne dépasse que 3 à 4 niveaux ; le B-tree est donc beaucoup plus plat. La recherche est donc plus optimal, on exclue rapidement beaucoup plus de données.

   À chaque descente dans l'arbre, la recherche dans une feuille peut être codée soit via une recherche dichotomique $O(\log_2 m) = O(\log_2 50) = O(1)$, soit via une recherche par interpolation $O(\log \log m) = O(\log \log 50) = O(1)$, cf. question 1.2. Car les données sont tries dans les feuilles. Comme on peut le constater, après simplification de l'analyse asymptotique, les deux recherches sont en $O(1)$ car le paramètre m est une constante. Cependant, en pratique, ce paramètre a une importance notable. Si les données insérées sont uniformes, alors les clés dans les feuilles sont (asymptotiquement) uniformément distribuées, dans ce cas la recherche par interpolation est plus intéressante. Dit autrement la distribution des clés dans un B-tree reflète la distribution des données d'entrée. (+1 pt)
3. Les fusions et emprunts (borrow) sont nécessaires lors des suppressions pour maintenir la propriété de remplissage minimal des nœuds ($\lceil m/2 \rceil - 1$ clés minimum, sauf racine). Sans cela, un nœud pourrait devenir trop vide, violant les invariants du B-arbre. Le principe est similaire pour les splits, cela permet de respecter les invariants.
4. Preuve de la hauteur : Dans un B-arbre d'ordre m, un nœud non-racine a au moins $\lceil m/2 \rceil - 1$ clés, donc au moins $\lceil m/2 \rceil$ fils. Le nombre minimal de clés à la hauteur h est supérieur ou égal à $2 \times (\lceil m/2 \rceil)^{h-1} - 1$. Donc $n + 1 \geq 2 \times (\lceil m/2 \rceil)^{h-1}$, ce qui donne après manipulation :

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

   Version longue et plus simple page suivante (+1) :

Cas de référence : BST équilibré Dans un arbre binaire de recherche équilibré, chaque nœud a au plus deux fils. Le nombre de nœuds par niveau double donc à chaque hauteur $1, 2, 4, \ldots, 2^h$. Le nombre total de nœuds vérifie alors $n \leq 2^h - 1$. Donc $n + 1 \leq 2^h \quad \Rightarrow \quad h \geq \log_2(n + 1)$.

Cas d'un B-arbre d'ordre m Dans un B-arbre d'ordre m : la racine a au moins 2 fils, tout autre nœud interne a au moins $\lceil m/2 \rceil$ fils. On considère le cas minimal (arbre le plus bas possible). Pour determiner le nombre minimal de nœuds par niveau, on obtient la croissance suivante :

— niveau 0 : 1
— niveau 1 : 2
— niveau 2 : $2 \cdot \lceil m/2 \rceil$
— niveau 3 : $2 \cdot (\lceil m/2 \rceil)^2$
— niveau h : $2 \cdot (\lceil m/2 \rceil)^{h-1}$

Ainsi, le nombre de clés vérifie $n \geq 2 \cdot (\lceil m/2 \rceil)^{h-1} - 1$. A partir de cette équation on cherche a définir h :

$$n + 1 \geq 2 \cdot (\lceil m/2 \rceil)^{h-1} \implies \frac{n+1}{2} \geq (\lceil m/2 \rceil)^{h-1}$$

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

On retrouve une généralisation directe du cas du BST $h = O(\log_{\lceil m/2 \rceil} n)$ ce qui montre que la hauteur d'un B-arbre croît logarithmiquement en fonction du nombre de clés.

6 Analyse et débogage – Tour de Hanoï itérative (5,0 points)

1. Complexité temporelle : $O(2^n)$ (le nombre de mouvements nécessaires est $2^n - 1$, et l'algorithme simule essentiellement tous les mouvements via la pile).
2. Complexité spatiale : $O(2^n)$ dans le pire cas (la pile peut contenir jusqu'à $O(2^n)$ tâches, car chaque appel récursif est simulé par un push).
3. Erreurs graves liées à la gestion de la mémoire (au moins 4) :
   (a) Pas de vérification du retour de `malloc` (risque de déréférencement de NULL).
   (b) Pas de vérification du retour de `realloc` : si `realloc` échoue, on perd l'ancien pointeur `stack` → fuite mémoire + plantage.
   (c) Aucune libération de la mémoire (`free(stack)`) à la fin de la fonction → fuite mémoire systématique.
   (d) Après un `realloc` réussi, on continue à utiliser `stack[top]` alors que le pointeur `stack` a potentiellement changé ; on devrait réaffecter correctement, mais surtout on risque d'utiliser une adresse invalide si realloc a déplacé le bloc.
   (e) (bonus) La logique des phases est incomplète : il manque le mouvement réel du disque lorsque `phase == 2` (on ne fait que décrémenter sans jamais afficher/déplacer le disque de taille 1).