A2025 – INFO I MPI

**ÉCOLE NATIONALE DES PONTS et CHAUSSÉES,
ISAE-SUPAERO, ENSTA PARIS,
TÉLÉCOM PARIS, MINES PARIS,
MINES SAINT-ÉTIENNE, MINES NANCY,
IMT ATLANTIQUE, ENSAE PARIS,
CHIMIE PARISTECH - PSL.**

**Concours Mines-Télécom,
Concours Centrale-Supélec (Cycle International).**

CONCOURS 2025

PREMIÈRE ÉPREUVE D'INFORMATIQUE

Durée de l'épreuve : 3 heures

L'usage de la calculatrice ou de tout dispositif électronique est interdit.

*Les candidats sont priés de mentionner de façon apparente
sur la première page de la copie :*

INFORMATIQUE I - MPI

Cette épreuve concerne uniquement les candidats de la filière MPI.

L'énoncé de cette épreuve comporte 13 pages de texte.

Si, au cours de l'épreuve, un candidat repère ce qui lui semble être une erreur d'énoncé, il le signale sur sa copie et poursuit sa composition en expliquant les raisons des initiatives qu'il est amené à prendre.

.

Étude des tableaux associatifs

Préliminaires

L'épreuve est composée d'un problème unique, comportant 34 questions. Le problème consiste en l'étude et l'implémentation d'un tableau associatif en C, suivant différentes méthodes.

Le problème est divisé en 4 sections distinctes :

— l'implémentation du tableau associatif en utilisant une structure d'arbre (section 1),

— l'ajout d'un « ramasse miettes » pour gérer la mémoire (section 2),

— l'implémentation du tableau associatif en utilisant un algorithme probabiliste (section 3),

— la conception d'un analyseur syntaxique pour créer un tableau associatif à partir d'une chaîne de caractères (section 4).

Les quatre sections sont indépendantes.

Travail attendu

Les résultats attendus pour certaines questions pourront être admis afin de faciliter la résolution des questions suivantes.

Le langage C constitue l'unique langage autorisé pour le traitement des questions de programmation.

Une attention particulière sera portée quant à la correction et la lisibilité du code proposé. Si une tolérance raisonnable peut être envisagée sur certains aspects syntaxiques dans le cadre d'une épreuve réalisée sur support papier (par exemple, l'oubli d'un « ; »), un code difficilement lisible ou l'emploi de fonctions ou structures de contrôle non conformes aux standards du langage C, notamment celles empruntées à d'autres langages, sera sanctionné.

Un soin particulier sera également apporté à la gestion de la mémoire, à la bonne initialisation des structures ainsi qu'au contrôle systématique des allocations mémoires.

La gestion rigoureuse des erreurs sera valorisée (l'utilisation d'`assert` est recommandée), de même que la qualité, la simplicité et la lisibilité du code.

En conformité avec le programme officiel, il sera admis que certains en-têtes standards soient présupposés comme inclus dans le code : `<assert.h>`, `<stdbool.h>`, `<stddef.h>`, `<stdint.h>`, `<stdio.h>`, `<stdlib.h>`, et `<string.h>`.

Les tableaux associatifs

Les tableaux associatifs sont des structures de données permettant un accès rapide à une *valeur* à partir d'une *clef*.

Afin de simplifier l'écriture du code, nous considèrerons que les clefs sont toujours de type `char*` et les valeurs de type `int`.

1. Implémentation par un arbre binaire de recherche

Le but de cette section est d'étudier une implémentation d'un tableau associatif reposant sur un arbre binaire de recherche.

Nous adopterons une approche fonctionnelle pour l'implémentation des structures de données, en les considérant comme immuables. Ainsi, chaque fonction de manipulation renverra systématiquement une nouvelle version modifiée de la structure, sans altérer l'originale.

Cette méthode présente l'avantage d'éliminer les effets de bord lors des appels de fonction. Toutefois, elle implique une augmentation du nombre d'allocations mémoire, puisque chaque modification nécessite la création d'une copie de la structure de données.

Dans cette section nous ne nous occuperons pas de la récupération de la mémoire, car nous introduirons en section 2 un « ramasse miettes » pour gérer la mémoire automatiquement.

Arbre binaire de recherche

Définitions : Soit A un arbre binaire de recherche. On notera avec une lettre minuscule (par exemple x ou y) un nœud de A, et avec une lettre majuscule (par exemple T_1, T_2 ou T_3) un sous-arbre de A. Soit x un nœud, on notera x_{gauche} et x_{droit}, respectivement le sous-arbre gauche et le sous-arbre droit de x.

On rappelle que la taille d'un arbre est égale au nombre de nœuds qu'il contient.

Pour un arbre de racine x, on notera $h(x)$ sa hauteur, et on définit récursivement la hauteur d'un arbre par :

— la hauteur d'un arbre vide est -1,

— $h(x) = \max(h(x_{\text{gauche}}), h(x_{\text{droit}})) + 1$.

Pour un arbre de racine x, on définit la valeur d'équilibre par :

$$e(x) = h(x_{\text{gauche}}) - h(x_{\text{droit}})$$

On qualifiera d'*équilibré* un arbre binaire de recherche A si pour tout nœud x de A,

$$-1 \leqslant e(x) \leqslant 1$$

Définition : On définit un nœud d'un arbre binaire de recherche par la structure suivante :

```
1.  struct abrNoeud_s {
2.    char* clef;
3.    int valeur;
4.
5.    struct abrNoeud_s* fils_gauche;
6.    struct abrNoeud_s* fils_droit;
7.  };
8.  typedef struct abrNoeud_s abrNoeud;
```

On utilisera la valeur du champ `clef` pour ordonner les nœuds de l'arbre, en utilisant l'ordre alphabétique.

☐ 1 – Nous nous intéressons aux valeurs du champ `clef` de la structure ci-dessus. Expliquer comment une chaîne de caractères est représentée en mémoire en C ? Combien d'octets sont nécessaires pour représenter la chaîne `"arbre"` par exemple ?

☐ 2 – Sachant que le code ASCII du caractère `'a'` est 97, quelle est sa représentation en binaire ? Quelle est sa représentation en hexadécimal ?

☐ 3 – Écrire une fonction C `copie_chaine` qui renvoie une copie de la chaîne de caractères passée en paramètre. La signature de la fonction est :

```
char* copie_chaine(char* chaine)
```

Indication C : On considèrera que la chaîne de caractères passée en paramètre (`chaine`) n'est jamais `NULL`.

☐ 4 – Écrire une fonction C `abr_creer_noeud` qui crée un nœud à partir d'une clef, d'une valeur et de 2 sous-arbres donnés, et qui renvoie ce nouveau nœud créé. La signature de la fonction est :

```
abrNoeud* abr_creer_noeud(char* clef, int valeur,
                          abrNoeud* fils_gauche,
                          abrNoeud* fils_droit)
```

Indication C : le champ `clef` du nœud créé doit contenir une copie de la chaîne de caractères passée en paramètre. Par contre, il n'est pas nécessaire de dupliquer les sous-arbres passés en paramètre.

☐ 5 – Écrire une fonction C `abr_equilibre` qui renvoie la valeur d'équilibre de l'arbre dont le nœud racine est passé en paramètre. La signature de la fonction est :

```
int abr_equilibre(abrNoeud* noeud)
```

☐ 6 – Pour un arbre de taille N, quelle est la complexité de la fonction `abr_equilibre` ? Sans proposer de code, expliquer quelles seraient les modifications à apporter à la structure de donnée et/ou à l'algorithme pour améliorer cette complexité ?

☐ 7 – Soit A un arbre binaire de recherche équilibré et de hauteur h, quelle approximation peut-on faire du nombre minimal N de nœuds que peut contenir cet arbre ?

☐ 8 – En déduire une borne supérieure de la hauteur h d'un arbre binaire de recherche équilibré de taille N.

☐ 9 – Écrire une fonction C récursive `abr_rechercher` qui, étant donné un arbre dont le nœud racine est passé en paramètre ainsi qu'une clef, renvoie la valeur associée à cette dernière. La signature de la fonction est :

```
int abr_rechercher(abrNoeud* noeud, char* clef)
```

Indication : On considèrera que la clef donnée est toujours dans l'arbre binaire de recherche.

Indication C : l'opérateur == permet de comparer des entiers ou des pointeurs mais pas de comparer le contenu de deux chaînes de caractères.

☐ 10 – En application de la question 8, quelle est la complexité de la fonction `abr_rechercher` pour un arbre équilibré de taille N ?

☐ 11 – Quel est l'intérêt d'avoir un arbre binaire de recherche qui soit tout le temps équilibré ?

2. Ajout d'un ramasse miettes

Un « ramasse miettes » est un ensemble de fonctions qui permet de gérer la mémoire de manière automatique. L'idée est de libérer la mémoire qui avait été allouée dynamiquement pour des structures ou des tableaux lorsque ces derniers ne sont plus utilisés.

Dans un contexte fonctionnel où les données sont immuables, il est parfois compliqué de déterminer le moment opportun pour libérer la mémoire allouée aux nœuds. En effet, l'immuabilité des structures permet de partager le même sous-arbre à plusieurs endroits d'un arbre, sans indication explicite de ce partage, rendant ainsi difficile de savoir si un nœud doit être libéré ou non. L'utilisation d'un ramasse miettes dans un tel cas est donc tout indiquée. C'est ce que nous allons étudier dans cette section.

Pour simplifier le problème, nous considérons qu'une structure ou un tableau devient inutilisé dès lors qu'il n'est plus atteignable à partir des variables globales du programme. Autrement dit, en partant des variables globales, il est impossible d'y accéder en suivant des pointeurs, même de manière transitive.

Pour pouvoir décrire l'algorithme de ramasse miettes qui nous intéresse, nous avons besoin de considérer qu'un nœud peut être « marqué ». Cela consiste à mémoriser une information dans la structure elle-même. Dans notre situation, nous utiliserons la structure suivante :

```
1.  struct abrNoeud_s {
2.    char* clef;
3.    int valeur;
4.
5.    struct abrNoeud_s* fils_gauche;
6.    struct abrNoeud_s* fils_droit;
7.
8.    // partie propre au ramasse-miettes
9.    bool marque;
10. };
11. typedef struct abrNoeud_s abrNoeud;
```

Un nœud est alors dit « marqué » si et seulement si le champ `marque` a pour valeur `true`. Nous considérons qu'initialement aucun nœud n'est marqué. L'algorithme de ramasse miettes fonctionne en deux temps :

1. Marquer tous les nœuds atteignables par clôture transitive depuis les variables globales. C'est-à-dire tous les nœuds atteignables directement ou bien atteignables à partir d'un nœud lui-même atteignable.
2. Libérer tous les nœuds alloués mais non marqués et effacer la marque des nœuds marqués.

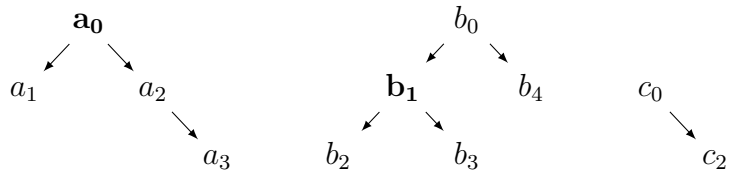

FIGURE 1 – Exemples d'arbres binaires de recherche, où les a_i, b_i, c_i sont des nœuds.

☐ 12 – On considère que *seuls* a_0 et b_1 sont directement référencés par des variables globales (Figure 1). Quels sont les nœuds marqués par l'étape 1 de l'algorithme ? Quels nœuds seront libérés par le ramasse miettes ?
En déduire (sans démonstration) la complexité en temps de l'algorithme de marquage et de l'algorithme de libération.

Pour pouvoir libérer tous les nœuds non marqués il est nécessaire de maintenir une liste des nœuds de l'arbre alloués en mémoire. Nous proposons d'utiliser la structure de données suivante :

```
1.  struct rmListeMemoire_s {
2.    abrNoeud* element;
3.    struct rmListeMemoire_s* suivant;
4.  };
5.  typedef struct rmListeMemoire_s rmListeMemoire;
```

☐ 13 – Écrire une fonction C permettant d'insérer, en temps constant, un nœud d'arbre binaire de recherche dans une liste. La signature de la fonction est :

```
rmListeMemoire* rm_inserer_tete(abrNoeud* element,
                                rmListeMemoire* liste)
```

Indication : la donnée référencée par `liste` n'a pas besoin d'être modifiée.

Nous considérons maintenant la structure de donnée **`RM`** permettant de mémoriser dans le champ `arbres` les nœuds racines des différents arbres binaires de recherche accessibles (dans notre cas il peut y avoir jusqu'à 16 nœuds racines) ainsi que la liste des nœuds alloués :

```
1.  #define MAX_NB_ARBRES 16
2.  struct RM_s {
3.    abrNoeud* arbres[MAX_NB_ARBRES]; /* arbres accessibles */
4.    int nb_elements;     /* nombre total de noeuds alloués */
5.    rmListeMemoire* premierElement;
6.  };
7.  typedef struct RM_s RM;
8.
9.  RM* rm; /* variable globale pour l'ensemble du programme */
```

Le champ `premierElement` est une référence vers le premier élément d'une liste. Cette dernière est gérée de la façon suivante :

— lorsqu'un nœud d'un arbre binaire de recherche est alloué, il est ajouté à la liste,

— lorsqu'un nœud d'un arbre binaire de recherche est libéré, il est retiré de la liste.

☐ 14 – La fonction `main` commence par l'instruction `rm = rm_creer_rm();`
Écrire le code de la fonction `rm_creer_rm` permettant d'allouer et d'initialiser la structure de données qui est affectée à la variable `rm`. La signature de la fonction est :

```
RM* rm_creer_rm(void)
```

Le code suivant correspond aux deux étapes de l'algorithme de ramasse miettes :

```
1.  void rm_ramasse_miettes(RM* rm) {
2.    assert(rm != NULL);
3.    int nb_noeuds = rm->nb_elements;
4.    for (int i = 0; i < MAX_NB_ARBRES; i++) {
5.      if (rm->arbres[i] != NULL) {
6.        rm_marquer_elements_accessibles(rm->arbres[i]);
7.      }
8.    }
9.    rm_recuperer_elements_inaccessibles(rm);
10.   printf("noeuds␣collectes␣:␣%d,␣", nb_noeuds - rm->nb_elements);
11.   printf("noeuds␣encore␣accessibles␣:␣%d\n", rm->nb_elements);
12. }
```

☐ 15 – Écrire une fonction C correspondant à la première étape de l'algorithme : marquer un nœud donné ainsi que tous les nœuds atteignables à partir de ce dernier. La signature de la fonction est :

```
void rm_marquer_elements_accessibles(abrNoeud* element)
```

Pour libérer tous les nœuds non marqués on considère la fonction suivante :

```
1.  void rm_recuperer_elements_inaccessibles(RM* rm) {
2.    assert(rm != NULL);
3.    rmListeMemoire* objet = rm->premierElement;
4.    rmListeMemoire* objets_conserves = NULL;
5.
6.    while (objet != NULL) {
7.      if (!objet->element->marque) {
8.        /* Ce noeud n'a pas été marqué,
9.           on le retire de la liste et on le libère */
10.       rmListeMemoire* a_detruire = objet;
11.       objet = a_detruire->suivant;
12.       // PARTIE A
13.       // ici il faut libérer la mémoire
14.       // FIN PARTIE A
15.       rm->nb_elements--;
16.     } else {
17.       // PARTIE B
18.       /* Ce noeud a été marqué,
19.          on enlève la marque et on passe au noeud suivant */
20.       objet->element->marque = false;
21.       rmListeMemoire* a_garder = objet;
22.       objet = a_garder->suivant;
23.       // FIN PARTIE B
24.     }
25.   }
26.   rm->premierElement = objets_conserves;
27. }
```

L'algorithme est le suivant :

— on parcourt tous les éléments de la liste `premierElement`,

— si le nœud n'est pas marqué, il faut libérer la mémoire et enlever le nœud de la liste,

— si le nœud est marqué il faut enlever la marque et le laisser dans la liste.

☐ 16 – Compléter le code de la partie A de la fonction ci-dessus pour libérer la mémoire lorsque le nœud est marqué.
Indication : Seul le code entre *// PARTIE A* et *// FIN PARTIE A* est à compléter. Il n'est pas nécessaire de recopier le code autour.

☐ 17 – Quelle spécificité ont les nœuds de la liste référencée par le champ `premierElement` à la fin de l'exécution de la fonction ?

☐ 18 – Expliquer pourquoi le code proposé dans la partie B de la fonction ci-dessus n'est pas correct, et proposez une correction.
Indication : Seul le code entre *// PARTIE B* et *// FIN PARTIE B* est à compléter. Il n'est pas nécessaire de recopier le code autour.

3. Implémentation par l'utilisation d'un algorithme probabiliste

Nous allons nous intéresser dans cette section à une autre manière d'implémenter un tableau associatif, en utilisant un algorithme probabiliste : les « *skip lists* ».

3.1. Principe des skip lists

Une skip list est une liste chaînée optimisée pour accélérer la recherche d'un élément. Pour ce faire, certains nœuds stockent dans un tableau des pointeurs vers d'autres nœuds situés plus loin dans la liste. Chaque case du tableau correspond à un « niveau » et un pointeur de niveau i mène vers un nœud qui possède lui aussi un pointeur de niveau i.

Le niveau 0 contient l'ensemble des éléments, classés par ordre alphabétique en fonction de leur clef. Les niveaux supérieurs regroupent un sous-ensemble des éléments, de façon à espacer progressivement les nœuds. Par exemple, idéalement, le niveau 1 contiendrait un nœud sur deux, le niveau 2 un nœud sur quatre, et ainsi de suite (voir Figure 2).

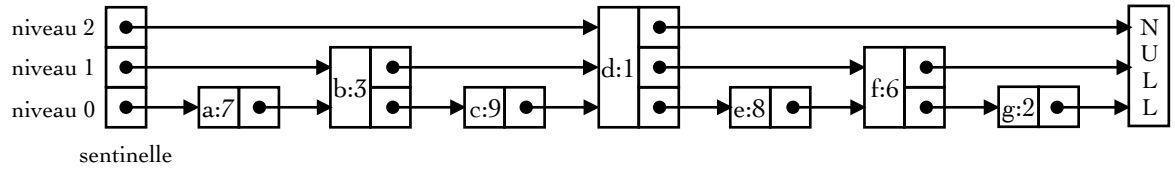

FIGURE 2 – Exemple d'une skip list idéale.

Le premier nœud de la skip list est une sentinelle : un élément spécial qui ne contient pas de donnée utile mais qui possède des pointeurs pour tous les niveaux de la liste.

Pour chercher un élément, on démarre à partir de la sentinelle au niveau le plus élevé. À ce niveau, on suit les pointeurs vers les nœuds suivants tant que la clef du nœud actuel est inférieure à celle recherchée. Dès qu'on rencontre un nœud dont la clef est supérieure, on descend d'un niveau et on répète la même opération. Lorsque la recherche atteint le niveau 0, elle est terminée et, si l'élément recherché y est présent, il est renvoyé (voir Figure 3).

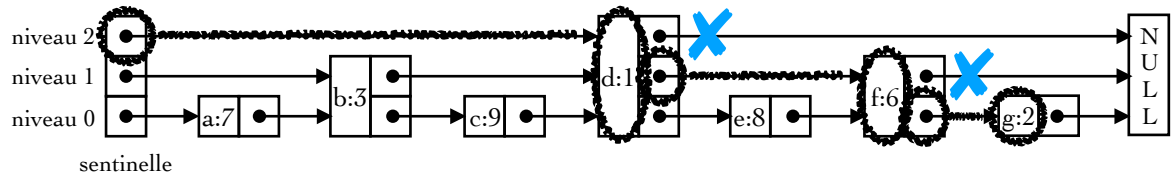

FIGURE 3 – Parcours lors de la recherche de la clef "g".

En pratique, chaque nœud a une probabilité de $\frac{1}{2}$ d'être présent dans le niveau supérieur.

3.2. Implémentation

Définition : On définit une skip list et un nœud d'une skip list par les structures suivantes :

```
1.  struct slNoeud_s {
2.    char* clef;
3.    int valeur;
4.
5.    struct slNoeud_s* suivant_par_niveau[];
6.  };
7.  typedef struct slNoeud_s slNoeud;
8.
9.  struct slListe_s {
10.   int niveau_actuel;
11.   slNoeud* sentinelle;
12. };
13. typedef struct slListe_s slListe;
```

On définit une constante, `MAX_NIVEAU`, qui correspond au nombre maximal de niveaux qu'un nœud peut avoir. La sentinelle de la skip list est créée comme un nœud particulier, avec une clef vide, une valeur nulle, et un tableau de pointeurs de taille `MAX_NIVEAU+1` (de 0 à `MAX_NIVEAU` inclus) tous initialisés à `NULL`.

☐ 19 – Écrire, en suivant l'algorithme décrit section 3.1, une fonction C `sl_rechercher` qui recherche, dans la skip list passée en paramètre, la clef donnée et qui renvoie :

— soit la valeur associée à la clef si elle est présente dans la skip list,

— soit -1 si la clef n'est pas présente dans la skip list.

La signature de la fonction est :

```
int sl_rechercher(slListe* liste, char* clef)
```

☐ 20 – On rappelle que la fonction `rand()` renvoie un entier aléatoire (de type **`int`**) compris entre 0 et un certain entier supérieur à 32767.

Quelles sont les propriétés de la valeur renvoyée par la fonction suivante ?

```
1.  int sl_mystere() {
2.    int niveau = 0;
3.    while ((rand() % 2) == 1 && niveau < MAX_NIVEAU) {
4.      niveau++;
5.    }
6.    return niveau;
7.  }
```

On suppose que la fonction suivante est disponible :

```
slNoeud* sl_creer_noeud(int niveau, char* clef, int valeur)
```

Cette fonction crée un nouveau nœud de niveau `niveau` avec la clef et la valeur passées en paramètre. Les pointeurs du nouveau nœud sont tous initialisés à `NULL`.

La fonction `sl_ajoute_valeur`, qui ajoute une clef et une valeur dans la skip list passée en paramètre, a la structure suivante :

```
1.  void sl_ajoute_valeur(slListe* liste, char* clef, int valeur) {
2.    int niveau = sl_mystere();
3.    slNoeud* nouveau_noeud = sl_creer_noeud(niveau, clef, valeur);
4.
5.    slNoeud* noeuds_a_mettre_a_jour[MAX_NIVEAU + 1];
6.    slNoeud* courant = liste->sentinelle;
7.
8.    // PARTIE A
9.    /* parcourt l'ensemble des niveaux de la liste pour trouver
10.      dans chaque niveau le noeud après lequel insérer
11.      le nouveau noeud */
12.   // FIN PARTIE A
13.
14.   if (niveau > liste->niveau_actuel) {
15.     // PARTIE B
16.     /* si le niveau du nouveau noeud est plus grand
17.        que le niveau actuel de la liste
18.        préparer la sentinelle */
19.     // FIN PARTIE B
20.   }
21.
22.   // PARTIE C
23.   /* insérer le nouveau noeud */
24.   // FIN PARTIE C
25. }
```

☐ 21 – En vous basant sur les instructions du commentaire, écrire le code correspondant à la partie A de la fonction `sl_ajoute_valeur`.

Indication C : Les nœuds devant être mis à jour sont stockés dans le tableau `noeuds_a_mettre_a_jour`.
Indication : Seul le code entre *// PARTIE A* et *// FIN PARTIE A* est à compléter. Il n'est pas nécessaire de recopier le code autour.

☐ 22 – En vous basant sur les instructions du commentaire, écrire le code correspondant à la partie B de la fonction `sl_ajoute_valeur`.

Indication C : On réutilisera la variable `noeuds_a_mettre_a_jour`.
Indication : Seul le code entre *// PARTIE B* et *// FIN PARTIE B* est à compléter. Il n'est pas nécessaire de recopier le code autour.

☐ 23 – En vous basant sur les instructions du commentaire, écrire le code correspondant à la partie C de la fonction `sl_ajoute_valeur`.

Indication : Seul le code entre *// PARTIE C* et *// FIN PARTIE C* est à compléter. Il n'est pas nécessaire de recopier le code autour.

☐ 24 – L'algorithme des skip lists est un algorithme probabiliste. Est-ce un algorithme de type « Monte-Carlo » ou de type « Las Vegas » ? Justifier.

☐ 25 – En supposant une skip list idéale comme dans la Figure 2, où chaque niveau possède exactement la moitié des éléments du niveau inférieur idéalement répartis, quelle est la complexité en temps de la recherche d'un élément dans une skip list de taille n ?

On considère désormais une skip list non idéale, où les éléments sont répartis de manière aléatoire avec une probabilité $p = \frac{1}{2}$ d'être présent dans le niveau supérieur.

☐ 26 – Quelle est l'espérance du nombre d'éléments au niveau k ?

☐ 27 – Quelle est l'espérance du niveau le plus haut d'une skip list de taille n ?

On acceptera dans la suite le résultat suivant : cette espérance est proportionnelle à $\log_2(n)$.

☐ 28 – Quel est en moyenne le nombre de nœuds parcourus horizontalement dans un niveau k lors d'une recherche dans une skip list de taille n ?

☐ 29 – Des questions 27 et 28, déduisez la complexité en moyenne en temps de la recherche d'un élément dans une skip list de taille n.

4. Analyse syntaxique

On souhaite construire une fonction `analyse` qui prend en paramètre une chaîne de caractères décrivant un tableau associatif sous la forme de couples `clef:valeur`.

Un exemple d'utilisation pourrait être : `analyse("{id:5,valeur:12,ok:0}")`

Le format de la chaîne de caractères passée en paramètre est spécifié par les règles suivantes :

- la chaîne doit commencer par une accolade ouvrante, être suivie d'une liste de couples `clef:valeur`, et terminée par une accolade fermante,
- chaque couple est composé d'une clef et d'une valeur séparées par le caractère '`:`',
- ni la clef ni la valeur ne peuvent être absentes,
- la clef est une suite finie non vide de lettres minuscules (de '`a`' à '`z`'),
- la valeur est un entier non signé, sans 0 non significatif,
- les couples sont séparés par le caractère '`,`'.

Le but de cette partie est d'étudier la grammaire et l'automate de ce langage de description.

Définition : On utilise la notation $x \mid \ldots \mid y$ pour indiquer la disjonction de symboles allant de x à y avec x et y des chiffres ou x et y des lettres.

On considère connues les règles de production suivantes :

$$\begin{array}{lcl} N & \to & 0 \mid \ldots \mid 9 \\ M & \to & 1 \mid \ldots \mid 9 \\ L & \to & a \mid \ldots \mid z \end{array}$$

☐ 30 – Écrire la règle de production `C` correspondant à la description d'une clef.

☐ 31 – Écrire la règle de production `V` correspondant à la description d'une valeur.

On considère la grammaire suivante :

$$\begin{array}{lcl} T & \to & \{S\} \\ S & \to & K \mid K, S \\ K & \to & C : V \end{array}$$

☐ 32 – Le mot `{id:5,valeur:12,ok:0}` appartient-il au langage engendré par cette grammaire ? Le cas échéant, dessinez un arbre de dérivation.
Même question pour le mot `{james:007}`.

☐ 33 – Décrire un automate (avec un schéma ou une table de transitions) déterministe et sans transition vide, correspondant au langage reconnu par la règle T. On s'autorisera à étiqueter les transitions avec des disjonctions de lettres ou de chiffres.

☐ 34 – Vérifiez en précisant les étapes de l'automate que les séquences `{code:102}`, `{v:0}`, et `{a:1,b:2}`, sont bien reconnues par l'automate proposé. Dans quel état est l'automate lorsqu'on lui demande de reconnaitre la séquence `{james:007}` ?

A. Rappels des signatures des fonctions de `<string.h>` en C

On rappelle les signatures et le fonctionnement de quelques fonctions de la bibliothèque standard du langage C, `string.h` :

`char* strcat(char* dest, char* src)`

Concatène la chaîne `src` à la fin de la chaîne `dest`. La chaîne `dest` doit être suffisamment grande pour contenir le résultat. La fonction renvoie un pointeur sur la chaîne `dest`.

`int strcmp(char* c, char* d)`

Compare les deux chaînes passées en paramètres. Elle renvoie un entier négatif si la première chaîne est inférieure à la seconde, 0 si les deux chaînes sont égales, et un entier positif si la première chaîne est supérieure à la seconde.

`char* strcpy(char* dest, char* src)`

Copie la chaîne `src` dans la chaîne `dest`. La chaîne `dest` doit être suffisamment grande pour contenir le résultat. La fonction renvoie un pointeur sur la chaîne `dest`.

`size_t strlen(char* c)`

Renvoie la longueur de la chaîne passée en paramètre, c'est-à-dire le nombre de caractères avant le caractère `'\0'` de fin de chaîne.

Fin de l'épreuve