

CONCOURS Mines Pont 2025

Informatique 1 — 3 heures

1 Implémentation par un arbre binaire de recherche

□ 1 — Une chaîne est un tableau de `char` (8 bits) suivi d'un marqueur de fin de chaîne `'\0'`. "arbre" possède 5 lettres et sera donc codé sur 6 octets.

□ 2 — $97 = 6 \times 16 + 1$, soit 61 en hexadécimal, donc 0110 0001 en binaire.

□ 3 — C'est plus facile avec l'annexe, non signalée...

```
1 char* copie_chaine(char* chaine){
2     dest = malloc(sizeof(char) * (strlen(chaine)+1));
3     assert(dest != NULL);
4     strcpy(chaine, dest);
5     return dest;
6 }
```

□ 4 — C'est une initialisation classique de structure :

```
1 abrNoeud* abr_creer_noeud(char* clef, int valeur,
2                             abrNoeud* fils_gauche,
3                             abrNoeud* fils_droit){
4     noeud = malloc(sizeof(abrNoeud));
5     assert(noeud != NULL);
6     noeud->clef = copie_chaine(clef);
7     noeud->valeur = valeur;
8     noeud->fils_gauche = fils_gauche;
9     noeud->fils_droit = fils_droit;
10    return noeud;
11 }
```

□ 5 — Une fonction hauteur est implémenté à cette fin; l'arbre vide à une valeur d'équilibre considérée nulle.

```
1 int hauteur(abrNoeud* noeud){
2     if (noeud == NULL) return -1;
3     int hg = hauteur(noeud->fils_gauche);
4     int hd = hauteur(noeud->fils_droit);
5     return 1 + ( hg>hd ? hg : hd);
6 }
7
8 int abr_equilibre(abrNoeud* noeud){
9     if (noeud == NULL) return 0;
10    return hauteur(noeud->fils_gauche) - hauteur(noeud->fils_droit);
11 }
```

□ 6 — Le calcul de la hauteur nécessite le parcours de tous les nœuds, d'où une complexité $O(N)$. Une complexité $O(1)$ est possible en rajoutant un champ `hauteur` dans la structure `abrNoeud_s`; cela compliquera les procédures d'insertion et de suppression dans un arbre qui devront maintenir ce champ à jour.

□ 7 — N obéit à la récurrence suivante :

$$\begin{aligned} N(-1) &= 0 \\ N(0) &= 1 \\ N(h) &= N(h-1) + N(h-2) + 1 \text{ pour } h > 0. \end{aligned}$$

Soit encore, en posant $M = N + 1$, $M(h) = M(h-1) + M(h-2)$ pour $h > 0$.

$M(h)$ est une suite récurrente linéaire d'ordre 2. En notant $\phi > 1$ et $-1 < \psi < 0$ les deux racines de $r^2 = r + 1$, la solution est de la forme : $N + 1 = A\phi^h + B\psi^h$.

Puisque $A\phi^h$ domine rapidement, conjecturons puis montrons que $N + 1 > \phi^h$.

— Initialisation : $N(-1) + 1 = 1 > \phi^{-1} \simeq 0,618$; $N(0) + 1 = 2 > 1$.

— Hérédité : $N(h) + 1 = N(h-1) + 1 + N(h-2) + 1 > \phi^{h-2}(\phi + 1) = \phi^h$, car $\phi + 1 = \phi^2$.

Donc pour tout arbre équilibré,

$$N > \phi^h - 1.$$

□ 8 — Il en découle immédiatement que $h < \log_\phi(N + 1)$, et même $h \leq \log_\phi(N)$ si $N > 0$, car h est entier : l'arbre possède une hauteur logarithmique en sa taille.

□ 9 — Aucune vérification n'est nécessaire ici, d'après la précondition sur la présence de la clé.

```

1  int abr_rechercher(abrNoeud* noeud, char* clef){
2      int cmp = strcmp(noeud->clef, clef);
3      if  cmp > 0 return abr_rechercher(noeud->fils_gauche, clef);
4      if  cmp < 0 return abr_rechercher(noeud->fils_droit, clef);
5      return noeud->valeur;

```

□ 10 — Une seule branche de l'arbre est parcourue, d'où une complexité logarithmique.

□ 11 — L'intérêt d'avoir un arbre binaire de recherche qui soit tout le temps équilibré est de garantir un accès rapide, en un temps *quasi constant* (logarithmique).

2 Ajout d'un ramasse miettes

□ 12 — Les nœuds marqués seront : $a_0, a_1, a_2, a_3, b_1, b_2, b_3$.

□ 13 — C'est une insertion classique en tête.

```

1  rmListeMemoire* rm_inserer_tete(abrNoeud* element,
2      rmListeMemoire* liste){
3      rmListeMemoire* maillon = malloc(sizeof(rmListeMemoire));
4      assert (maillon != NULL);
5      maillon->element = element;

```

```

6     maillon->suivant = liste;
7     return maillon;
8 }

```

□ 14 — Tout devant être initialisé à 0 ou NULL, un `calloc` (pas plus au programme MPI que `#define MAX_NB_ARBRES 16`) est efficace ici. Il n’y a pas besoin d’allouer `arbres` qui est pré-alloué.

```

1 RM* rm_creer_rm(void){
2     RM* rm = calloc(1,sizeof(RM));
3     return rm;
4 }

```

□ 15 — Il faut parcourir récursivement les nœuds accessibles, en utilisant et propageant la précondition `element != NULL`.

```

1 void rm_marquer_elements_accessibles(abrNoeud* element){
2     element->marque = true;
3     if (element->fils_gauche != NULL)
4         rm_marquer_elements_accessibles(element->fils_gauche);
5     if (element->fils_droit != NULL)
6         rm_marquer_elements_accessibles(element->fils_droit);
7 }
8 }

```

□ 16 — Il y a une coquille d’énoncé, il faut bien sur détruire les éléments non marqués : pour cela il faut ajouter `free(a_detruire->element); free(a_detruire);` en ligne 13.

□ 17 — Les éléments pointés sont les éléments accessibles, et ils ont été démarqués en vue du passage suivant (après la correction de la question suivante, sinon `premierElement` vaut NULL et il n’y rien à dire).

□ 18 — Il manque `rm_inserer_entete(objet,objet_conserve);` en ligne 20.

3 Implémentation par l’utilisation d’un algorithme probabliste

□ 19 — Pour chaque niveau (`for`), L’algorithme avance autant que possible (`while`). `niveau_actuel` est le niveau de départ dans la sentinelle, qui correspond au plus haut nœud actuel. La présence éventuelle de la clé est vérifiée en fin de parcours.

```

1 sl_rechercher(slListe* liste, char* clef) {
2     slNoeud* noeud = liste->sentinelle;
3     for (int niv = liste->niveau_actuel; niv >= 0; niv--) {
4         while (noeud != NULL) {
5             slNoeud* suivant = noeud->suivant_par_niveau[niv];
6             if (suivant != NULL && strcmp(suivant->clef, clef) < 0)
7                 noeud = suivant; // on avance

```

```

8         else break;
9     } // on descend
10 }
11 if (noeud != NULL && strcmp(suivant->clef, clef) == 0)
12     return suivant->valeur; // on a trouvé
13 return -1;
14 }

```

□ **20** — La fonction `sl_mystere` renvoie n avec une probabilité 2^{-1-n} , sauf si $n = \text{MAX_NIVEAU}$ pour lequel elle vaut 2^{-n} ; ainsi la somme des probabilités vaut bien 1. Cela permet d'approximer la distribution idéale présentée en **3.1**.

□ **21** — (PARTIE A) : C'est le même parcours de `sl_rechercher`, avec les changements suivants :

- la clé n'est pas présente d'après l'énoncé, donc non cherchée.
- les nœuds parcourus sont stockés à la fin de chaque boucle;
- il suffit optimalement de démarrer au niveau `max(niveau, liste->niveau_actuel)`. Mais ceci semble être constaté qu'en partie B; Le niveau de départ est pris à `MAX_NIVEAU` pour simplifier, ce que semble indiquer le commentaire « on parcourt l'ensemble des niveaux de la liste ».

Ce qui donne :

```

1 slNoeud* noeud = liste->sentinelle;
2 for (int niv = MAX_NIVEAU; niv >= 0; niv--) {
3     while (noeud != NULL) {
4         slNoeud* suivant = noeud->suivant_par_niveau[niv];
5         if (suivant != NULL && strcmp(suivant->clef, clef) < 0);
6             noeud = suivant; // on avance
7         else break;
8     } // on descend
9     noeuds_a_mettre_a_jour[niv] = noeud;
10 }

```

□ **22** — La mise à jour de la sentinelle de la sentinelle (PARTIE B) se résume à `liste->niveau_actuel = niveau`; ici.

□ **23** — Pour la mise à jour des pointeurs (PARTIE C), il faut démarrer `niveau` : les autres pointeurs ne voient pas le nouveau nœud.

```

1 for (int niv = niveau; niv >= 0; niv--) {
2     nouveau_noeud->suivant_par_niveau[niv] =
3         noeuds_a_mettre_a_jour[niv]->suivant_par_niveau[niv];
4     noeuds_a_mettre_a_jour[niv]->suivant_par_niveau[niv] = nouveau_noeud;
5 }

```

□ **24** — C'est un algorithme « Las Vegas » : Le résultat final est exact.

□ **25** — C'est exactement le principe d'une recherche dichotomique, l'espace de recherche est divisé par deux à chaque comparaison de clé : la recherche se fait en temps logarithmique.

□ **26** — Soit N_k la variable aléatoire correspondant au nombre d'éléments au niveau k , et n le nombre de nœud de la skip list, $E(N_0) = n$ et $E(N_k) = \frac{1}{2}E(N_{k-1})$ pour $k > 0$. Soit par récurrence immédiate :

$$E(N_k) = \frac{n}{2^k}.$$

□ **27** — Il vaut mieux admettre le résultat. Mais bon : montrons que $E_n \sim_{\infty} \log_2(n)$.

Soit $n > 0$, et M_n la variable aléatoire correspondant au niveau maximum d'une skip list de taille n . La probabilité qu'un nœud donné atteigne le niveau k est 2^{-k} , donc celle de ne pas l'atteindre est $1 - 2^{-k}$. Il s'en suit que $p(M_n < k) = (1 - 2^{-k})^n$, et donc enfin que

$$p(M_n \geq k) = 1 - (1 - 2^{-k})^n \stackrel{\text{def}}{=} f_n(k).$$

d'autre part,

$$E(M_n) = \sum_{k=0}^{+\infty} k p(M_n = k) = \sum_{k=1}^{+\infty} p(M_n \geq k) = \sum_{k=1}^{+\infty} f_n(k).$$

comme $f_n(x)$ est décroissante sur $\mathbb{R}^+$:

$$\int_1^{+\infty} f_n(x) dx \leq \sum_{k=1}^{+\infty} f_n(k) \leq \int_0^{+\infty} f_n(x) dx.$$

Avec le changement de variable $t = 1 - 2^{-x} = 1 - e^{-ax}$ où $a = \ln(2)$, il vient $dt = a(1 - t) dx$, puis :

$$\int_0^{+\infty} f_n(x) dx = \frac{1}{a} \int_0^1 \frac{1 - t^n}{1 - t} dt = \frac{1}{a} \sum_{k=0}^{n-1} \int_0^1 t^k dt = \frac{1}{a} \sum_{k=1}^n \frac{1}{k} \stackrel{\text{def}}{=} A_n.$$

Enfin $\int_0^{+\infty} f_n(x) dx \leq 1 + \int_1^{+\infty} f_n(x) dx$ car $f_n(x) \leq 1$. D'où $A_n - 1 \leq E(M_n) \leq A_n$, soit finalement :

$$E_n \sim_{\infty} \frac{\ln(n)}{\ln(2)} = \log_2(n).$$

□ **28** — À un niveau donné, il y a une chance sur deux pour que la valeur cherchée soit inférieure ou supérieure à celle du nœud suivant. Soit en moyenne $\frac{1}{2}$ nœud parcouru à un niveau donné.

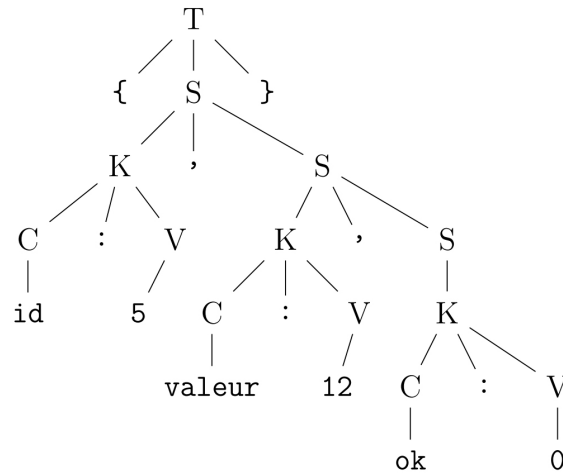
□ **29** — Comme le niveau maximal est de l'ordre de $\log_2(n)$ (**Q 27.**), l'algorithme réalise de l'ordre de $\log_2(n)$ déplacements verticaux et de l'ordre de $\frac{\log_2(n)}{2}$ déplacements horizontaux (**Q 28.**). D'où une complexité temporelle de recherche logarithmique. C'est la même complexité que la recherche dans un ABR.

4 Analyse syntaxique

□ **30** — Pour produire une clef : $C \rightarrow L|CL$.

□ **31** — Pour produire une valeur, par exemple : $V \rightarrow M|NF$, avec $F \rightarrow M|MF$.

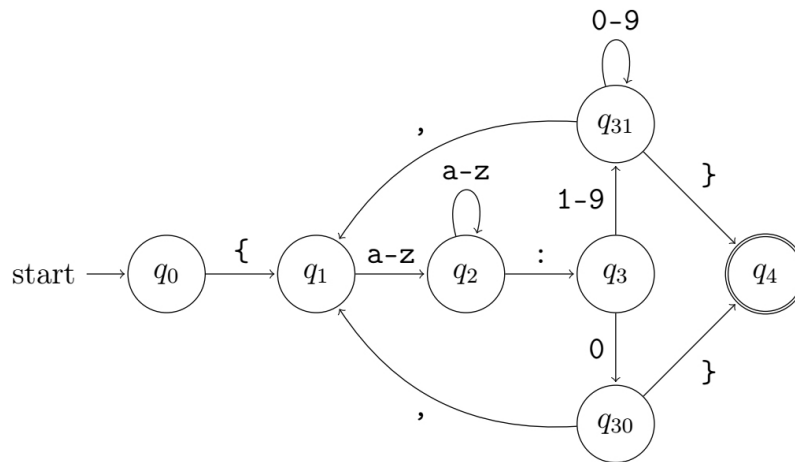
□ **32** — Le mot $\{\text{id}:5,\text{valeur}:12,\text{ok}:0\}$ possède pour arbre de dérivation :



Les règles C et V pour les feuilles ont été « contractées ».

Le mot $\{\text{james}:007\}$ n'est par contre pas dans le langage, car $007 \notin \mathcal{L}(V)$.

□ **33** — En notant $\mathbf{a-z}$ pour $\mathbf{a} \mid \dots \mid \mathbf{z}$, l'automate suivant convient :



□ **34** —

- $\{\text{code}:102\}$ correspond au chemin $q_0 \rightarrow q_1(\rightarrow q_2)^4 \rightarrow q_3(\rightarrow q_{31})^3 \rightarrow q_4$.
- $\{\text{v}:0\}$ correspond au chemin $q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow q_3 \rightarrow q_{30} \rightarrow q_4$.
- $\{\text{a}:1,\text{b}:2\}$ correspond au chemin $q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow q_3 \rightarrow q_{31} \rightarrow q_1 \rightarrow q_2 \rightarrow q_3 \rightarrow q_{31} \rightarrow q_4$.
- $\{\text{james}:007\}$ correspond au chemin $q_0 \rightarrow q_1(\rightarrow q_2)^5 \rightarrow q_3 \rightarrow q_{30}$. L'automate est alors dans un état bloqué car il n'est pas complet.