Quelques conseils

1. Lisez l'énoncé. Le sujet est difficile. Comprenez bien le but de la structure de donnée, et la façon dont elle fonctionne.
2. Il y a trois type de noeuds. Sur un brouillons, dessinez les trois exemples les plus simple possible utilisant ces noeuds, et faites tourner *à la main* les algorithmes sur ces exemples. Ça vous permet de réfléchir aux questions en divisant le problème en 3, et vous évite les erreurs les plus basiques.
3. Le sujet n'a rien dit sur la gestion des mauvais inputs, du coup, faites la supposition la plus confortable. Tous les inputs font sens et ne dépassent pas les bornes. Si vous avez fini le sujet, vous pourrez revenir pour ajouter des cas supplémentaires pour gérer les inputs illégaux. Le faire directement, c'est complexifier le code, et risquer de traiter moins de fonctions.
4. Le sujet propose beaucoup de code. Parcourez-le, lisez-le. Il est probable que vous puissiez copier-coller certaines constructions sans problème. Cela vous permet aussi de faire disparaitre les ambiguités qui pourraient rester après la lecture du sujet.
5. Gardez en tête qu'on peut ici parler de chaînes de caractères de plusieurs GB de taille mémoire. Copier les chaines ne doit être fait que si demandé explicitement, la gestion de la mémoire (`malloc/free`) doit être sans faille.

Question 1.1

Réussissez la première question. Elle influe énormément sur la perception que votre correcteur aura du reste de la copie (pour éviter ce biais, j'ai toutes les questions 1.1 de toutes les copies, puis, dans un ordre différent, toutes les questions 1.2, etc.).

Cette question a bien été traitée par 33% des candidats. La structure de cette function pouvait être récupérée directement depuis par exemple `get_left_child()`.

```
rope get_left_child(rope r){
  kind k = get_kind(r);
  if(k==concatenation){return r->conc.left;}
  else if (k==factor){return r->fac.node;}
  else{return r;}
}
```

La principale difficulté était de bien ajuster l'argument pos en fonction des situations, pour l'appel récursif. Voici une correction.

```
char char_at(rope ch, int pos){
  kind k = get_kind(ch);

  if(k==concatenation){
    int n = length(get_left_child(ch));

    if (pos < n){
      return char_at(get_left_child(ch), pos);
    }
    else {
      return char_at(get_right_child(ch), pos-n);
    }
  }

  else if(k==factor){
    return char_at(get_left_child(ch), pos + get_offset(ch));
  }

  else{
    return get_string(ch)[pos];
  }
}
```

Question 1.2

Il fallait bien comprendre ici que l'on désirait rajouter des noeuds à l'arbre sans copier les chaines de caractère sous-jascentes. L'indication, et la nécessité d'une complexité en temps constant forcaient la méthode, et notamment interdisaient une descente dans l'arbre, d'une complexité linéaire dans le pire des cas.

Pour traiter au mieux cette fonction, il fallait aussi traiter le cas `start == 0 && start + size == length(ch)`, traité par quelques-uns, mais de façon la plus souvent incorrecte : une rope vide n'est pas égale à `NULL`, mais à `of_string("")`.

Le cas `size == 0`, pourtant trivial, n'a été traité par une seule personne, incorrectement (`exit(EXIT_FAILURE);` n'est pas un comportement adapté).

```
rope delete(rope ch, int start, int size){
  if (size != 0) {

    if (start == 0) {
      if (start + size == length(ch)){ // we delete everything
        return of_string("");
      } else {
        return sub_rope(ch, size, length(ch) - size);
      }
    }

    else if (start + size == length(ch)) {
      return sub_rope(ch, 0, start);
    }

    else {
      return concat(sub_rope(ch, 0, start), sub_rope(ch, start+lg, length(ch) - size - start);
    }

  } else{
    return ch;
  }
}
```

Question 1.3

Le "moins de noeud possible" a été interprêté par certains comme une mise à plat de la chaine similaire à to_string. La contrainte du temps constant ne permettait pas cette approche. Les cas `pos = 0`, `pos == length(ch1)`, `length(ch1) == 0` et `length(ch2) == 0` n'ont jamais été tous traités.

```
rope insert(rope ch1, rope ch2, int pos){

  if (length(ch1) == 0) {
    return ch2;
  }
  else   if (length(ch2) == 0) {
    return ch1;
  }
  else if (pos == 0) {
    return concat(ch2, ch1)
  }
  else if (pos == length(ch1)) {
    return concat(ch1, ch2)
  }
  else {
    return concat(sub_rope(ch1, 0, pos), concat(ch2, sub_rope(ch1, pos, length(ch1) - pos)));
  }
}
```

Question 1.4

Ici, une approche récursive naïve suffisait et a été adoptée par nombreux d'entre vous, avec de monstrueuses fuites de mémoire pour tout le monde sauf une personne qui a fait des `free`.

Dans l'approche naïve, il faut faire attention en concaténant des chaines de caractère. `strcat(s1, s2)` ne peut être utilisé que si `s1` contient assez de place pour contenir aussi `s2`. The terminating null character `\0` has also to be correctly placed. Notons que l'utilisation de `strncat` est impérative pour les `factor`. Cela donne:

```
char *  to_string(rope ch){
  kind k = get_kind(ch);

  if(k==concatenation){
    char* left  = to_string(get_left_child(ch));
    char* right = to_string(get_right_child(ch));
    char * res = malloc(sizeof(char) * (length(ch) + 1));
    res[0]='\0'; // crucial for strcat.
    strcat(left, res)
    strcat(right, res)

    // we malloced only if the child is not a str
    if (get_kind(get_left_child(ch) != str)
        free(left);
    if (get_kind(get_right_child(ch) != str)
        free(right);
    return res;
  }

  else if(k==factor){
    char* s = to_string(get_left_child(ch));
    char * res = malloc(sizeof(char) * (length(ch) + 1));
    res[0]='\0'; // crucial for strcat.
    strncat(s[get_offset(ch)], res, length(ch));
    if (get_kind(get_left_child(ch) != str)
        free(s);
    return res;
  }

  else{
    return get_string(ch);
  }
}
```

L'approche que nous adoptons ici effectue beaucoup de copies. Dans le cas ou la rope est un arbre ne contenant que des concaténations avec à droite une feuille ne contenant qu'un caractère et à gauche une autre concaténation ou une feuille à un caractère, on va copier le plus profond caractère `r` fois, et donc obtenir une complexité en `O(r^2)`.

Question 1.5

Cette question était particulièrement difficile, personne n'a proposé de solution satisfaisante.

On peut faire du `O(r)` à la question précédente, et il faut créer une structure d'itérateur. Pour cela, on a besoin d'une structure de pile qui stocke des sub_rope de string qui, mises bout à bout, créent la string qu'il nous faut.

On fait une fonction pour écrire la pile, et, pour 1.4, une fonction pour la compiler en une chaine de caractère de la longueur de la rope globale. On évite ainsi de multiple copies.

Cette structure nous sert aussi pour la question 1.5. On peut en effet créer une structure qui garde la pile et un offset, et qui dépile au fur et à mesure, et rend le caractère courant, et incrémente l'offset à chaque appelle.

En utilisant deux structures pour chacune des ropes, on peut alors comparer caractère par caractère, en temps constant, sans créer les chaînes de caractère.

Question 2.1

Les deux première questions de la partie 2 était probablement les plus simple de l'énoncé, il suffisait de transcrire litéralement l'énoncé en code. Pourtant, moins de la moitié ont abordé ces questions. L'existence de parties dans un énoncé est souvent indicatrice d'indépendance. Vous pouvez abordez la partie 2 avant la partie 1.

Dans 2.1, if fallait utiliser `add_to_offset` et `set_sub_rope`. Solution de l'un de vos camarade:

```
void sub_sub_case(rope ch){

    rope chs = get_left_child(ch);

    add_to_offset(ch, get_offset(chs));
    set_sub_rope(ch, get_left_child(chs));

    destroy(chs);
}
```

Le destroy est une touche sympathique (et optionnelle).

Question 2.2

```
void sub_append_case(rope ch){

    int k, len, len_r1;
    rope son_ch, r1, r2, tmp;

    k = get_offset(ch);
    len = length(ch);
    son_ch = get_left_child(ch);
    r1 = get_left_child(son_ch);
    r2 = get_right_child(son_ch);
    len_r1 = length(r1);
    tmp = ch;

    if (k < len_r1 && (k+length(ch)) <= length(r1))
        ch = sub_rope(r1, k, len);
    else if (len_r1 <= k)
        ch = sub_rope(r2, k-len_r1, len);
    else {

        rope subg, subd;

        subg = sub_rope(r1, k, len_r1-k);
        subd = sub_rope(r2, 0, len+k-len_r1);

        turn_into_append(ch, subg, subd);
    }

    destroy(tmp);
}
```

Question 2.3

Cette question n'a été abordé par personne, mais était plus facile qu'il n'y paraissait. On sait maintenant transformer sub(sub()) en sub() et sub(conc()) en sub() ou en conc(sub()) on peut donc "pousser les sub vers les feuilles, en faisant remoneter les conc au-dessus des sub, et en supprimant les piles de sub.

Question 2.4

Une solution d'un de vos camarades. Il fallait bien penser à décrementer les sous-arbres.

```
void destroy(rope ch){

    if (ch == NULL) return;

    if (get_nbparents(ch) == 0) {

        kind k = get_kind(ch);

        if (k==concatenation) {
            rope ch1, ch2;

            ch1 = get_right_child(ch);
            ch2 = get_left_child(ch);

            dec_nbparents(ch1);
            dec_nbparents(ch2);

            destroy(ch1);
            destroy(ch2);
        }
        else {
            rope chi = get_left_child(ch);

            dec_nbparents(chi);

            destroy(chi);
        }

        free(ch);
    }
}
```

Question 2.5

Cette question n'a été abordé par personne.