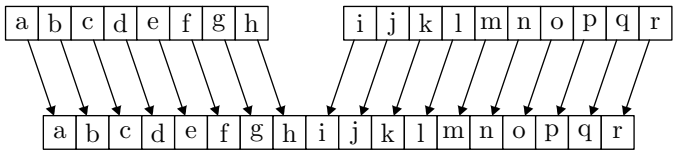
Licence Informatique **INF352 – Algorithmique 3**
Université Bordeaux 1

TP noté avril 2011

Lorsque l'on manipule des chaînes de caractères très longues, les opérations de concaténation et d'extraction de sous-chaînes peuvent s'avérer coûteuses. Par exemple, lorsque l'on veut concaténer deux chaînes de longueur k et l, il faut recopier dans une zone de mémoire de taille $k + l$ les $k + l$ caractères des deux chaînes.



Il en résulte que la concaténation nécessite :

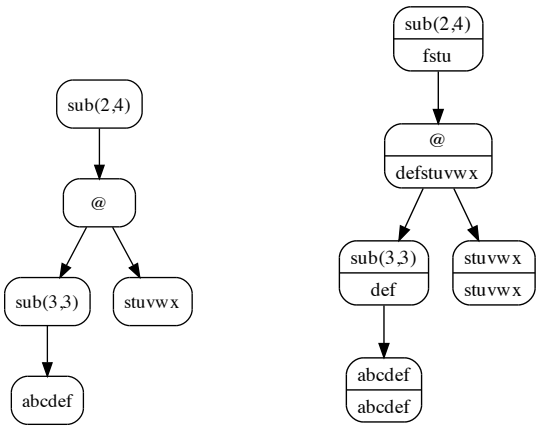
1. d'avoir à disposition en mémoire deux fois l'espace qui permet de contenir les deux chaînes à concaténer,
2. un temps de l'ordre de la taille des deux chaînes de caractères.

Dans le cas où les chaînes de caractères que l'on manipule sont très longues, tant les limitations de la mémoire que l'efficacité des opérations sur les chaînes peuvent se révéler rédhibitoires au bon fonctionnement du logiciel que l'on construit.

Les *cordes* sont une structure de données permettant de manipuler de façon efficace des chaînes de caractères très longues. Elles cherchent à remplir les spécifications suivantes :

1. elles doivent être persistantes, c'est-à-dire que si l'on construit une corde à partir d'autres cordes, ces dernières ne doivent pas être modifiées par un effet de bord,
2. les opérations de concaténation et d'extraction de sous-chaînes doivent être efficaces notamment en espace,
3. les chaînes de caractères représentées doivent pouvoir être de longueur arbitraire.

Les cordes sont représentées par des arbres utilisant trois types de nœuds : des nœuds binaires `conc`, des nœuds unaires $\texttt{sub}_{i,j}$ (où i et j sont des entiers positifs) et $\texttt{ch}_w$ (où w est une chaîne de caractères). On pourra écrire ses arbres par des expressions comme $\texttt{sub}_{2,4}(\texttt{conc}(\texttt{sub}_{3,3}(\texttt{ch}_{abcdef}), \texttt{ch}_{stuvw}))$ ou bien graphiquement par :



Chaque corde représente une chaîne de caractères (voir figure ci-dessus) : la corde $\texttt{conc}(t_1, t_2)$ représente la concaténation des chaînes représentées par t_1 et t_2, la corde $\texttt{sub}_{i,j}(t)$ représente la chaîne $a_i \dots a_{i+j-1}$ si la corde t représente la chaîne $a_0 \dots a_n$ et si i et $i + j - 1$ sont dans $[0; n]$, c'est à dire la sous-chaîne de $a_0 \dots a_n$ de longueur j commençant à la lettre d'indice i.

En vue de résoudre les exercices, vous disposez des fonctions suivantes qui sont déjà implémentée dans le fichier `rope.c` :

1. `char * new_string(int n)`, crée une chaîne de caractères de longueur n (les caractères composants cette chaîne peuvent être arbitraires).
2. `int length(rope r)` : renvoie la longueur de la chaîne telle qu'elle est stockée dans le nœud racine de r.
3. `kind get_kind(rope r)` : revoie une valeur dans le type

   ```
   enum kind{string, concatenation, factor}
   ```

   Si la fonction renvoie `string`, c'est que la corde `r` est de la forme $\texttt{ch}_w$; si elle renvoie `concatenation`, c'est que la corde `r` est de la forme $\texttt{conc}(r_1, r_2)$; et si elle renvoie `factor` , c'est que la corde `r` est de la forme $\texttt{sub}_{i,j}(r')$.
4. `char * get_string(rope r)` qui retourne la chaîne de caractères w si `r` est de la forme $\texttt{ch}_w$.
5. `int get_offset(rope r)` qui retourne l'entier i si `r` est de la forme $\texttt{sub}_{i,j}(r')$.
6. `rope concat(rope r1, rope r2)` qui crée la corde $\texttt{conc}(\texttt{r1}, \texttt{r2})$.
7. `rope sub_rope(rope r)` qui crée la corde $\texttt{sub}_{i,j}(\texttt{r})$,
8. `rope of_string(char * str)` qui crée la corde $\texttt{ch}_{\texttt{str}}$ à partir de la chaîne `str`.

9. `rope get_left_child(rope r)`, renvoie r_1 si `r` est de la forme $\texttt{conc}(r_1, r_2)$, renvoie r' si `r` est de la forme $\texttt{sub}_{i,j}(r')$, et renvoie r sinon.
10. `rope get_right_child(rope r)`, renvoie r_2 si `r` est de la forme $\texttt{conc}(r_1, r_2)$, renvoie r' si `r` est de la forme $\texttt{sub}_{i,j}(r')$, et renvoie `r` sinon.

Exercice 1

Question 1.1 Ecrire une fonction `char charat(rope r, int pos)` qui étant données une corde `r` représentant une chaîne de caractères $a_0 \dots a_n$ et un entier `pos` revoie la lettre $a_{\texttt{pos}}$.

Question 1.2 Ecrire une fonction

```
rope delete(rope r, int pos, int length)
```

qui étant donnés une corde `r` représentant une chaîne $a_0 \dots a_n$ et deux entiers i et j renvoie une corde r' qui représente la chaîne $a_0 \dots a_{\texttt{pos}-1} a_{\texttt{pos}+\texttt{length}} \dots a_n$ c'est-à-dire qui efface la sous-chaîne de `r` de longueur `lenght` commençant à l'indice `pos`. Cette fonction doit s'effectuer en temps constant et doit renvoyer une corde contenant le moins de nœuds possible.

Indication : on a seulement besoin de se servir des fonctions `rope concat(rope r1, rope r2)` et `rope sub_rope(rope r)` ; faites, entre autre, attention aux cas où $\texttt{pos} = 0$, $\texttt{pos} + \texttt{length} = n + 1$.

Question 1.3 Ecrire une fonction

```
rope insert(rope r1, rope r2, int pos)
```

qui étant donnés deux cordes r_1 et r_2 représentant respectivement les chaînes $a_1 \dots a_n$ et $b_1 \dots b_p$ revoie une corde représentant la chaîne

$$a_0 \dots a_{\texttt{pos}-1} b_1 \dots b_p a_{\texttt{pos}} \dots a_n$$

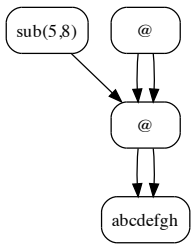
Cette fonction doit s'effectuer en temps constant et doit renvoyer la corde contenant le moins de nœuds possible.

Question 1.4 Ecrire une fonction `char * to_string(rope r)` qui renvoie la chaîne de caractères représentée par la corde r. On souhaite que chaque nœud de la structure ne soit traversé qu'au plus une fois (*i.e.* il s'agit donc de ne pas construire la chaîne en utilisant la fonction `char charat(rope r, int pos)`). Exprimer en fonction de `r` le temps nécessaire au calcul de cette fonction.

Question 1.5 Ecrire une fonction `int equal(rope r1, rope r2)` qui renvoie 1 si la chaîne de caractères représentée par `r1` est égales à celle représentée par `r2` et 0 sinon. On ne souhaite pas calculer ces chaînes explicitement pour vérifier cette égalité et on souhaite que chaque nœud de la structure soit traversé le moins de fois possible.

Un des intérêts des cordes est que l'on peut utiliser plusieurs fois une même corde pour en définir un plus grande. Il en résulte une utilisation efficace de la mémoire grâce au partage de structures. On peut ainsi partager les cordes qui sont copiées, mais également représenter des cordes avec moins de noeuds en remarquant des identités comme $\texttt{sub}_{k,l}(\texttt{sub}_{i,j}(r)) \equiv \texttt{sub}_{k+i,l}(r)$ (où la relation $r \equiv r'$ est vraie si et seulement si les cordes r et r' représentent les même chaînes).

Ainsi on peut instancier en mémoire une corde r représentant la chaîne $(abcdefgh)^4$ et la corde $\texttt{sub}_{5,8}(r)$ suivant la figure ci-dessous :



Cependant le partage de mémoire n'est pas sans présenter de problèmes. En effet, si l'on souhaite effacer une corde de la mémoire, il faut avant pouvoir s'assurer qu'elle n'est pas partagée. Aussi, nous allons considérer que chaque nœud de cordes contient un entier qui correspond au nombres de parents qu'il a dans d'autres cordes. On dispose ainsi des fonction suivantes :

1. `int get_nbparents(rope r)`(r) : renvoie le nombres de pointeurs qui référencent le nœud racine de la corde r,
2. `void dec_nbparents(rope r)` : décrémente le nombre de pointeurs qui référencent la corde r.

Les fonctions qui permettent de créer des nœuds à partir d'autres cordes, à savoir :

1. `rope concat(rope r1, rope r2)`,
2. `rope sub_rope(rope r)` et
3. `rope of_string(char * str)`,

créent des nœud dont le nombre de parents est 0, et elles incrémentent le nombre de parents des cordes éventuelles qu'elle prennent comme argument.

Dans l'exercice suivant, nous allons exploiter les identités suivantes :

$$
\begin{array}{rcl}
\texttt{sub}_{k,l}(\texttt{sub}_{i,j}(\texttt{r})) & \equiv & \texttt{sub}_{k+i,l}(\texttt{r}) \\
\texttt{sub}_{k,l}(\texttt{conc}(\texttt{r1},\texttt{r2})) & \equiv & \texttt{sub}_{k,l}(\texttt{r1}) \\
 & & \text{si } k < \texttt{length}(\texttt{r1}) \text{ et } k+l \leq \texttt{length}(\texttt{r1}) \\
\texttt{sub}_{k,l}(\texttt{conc}(\texttt{r1},\texttt{r2})) & \equiv & \texttt{sub}_{k-\texttt{length}(\texttt{r1}),l}(\texttt{r2}) \\
 & & \text{si } \texttt{length}(\texttt{r1}) \leq k \\
\texttt{sub}_{k,l}(\texttt{conc}(\texttt{r1},\texttt{r2})) & \equiv & \texttt{conc}(\texttt{sub}_{k,\texttt{length}(\texttt{r1})-k}(\texttt{r1}), \texttt{sub}_{0,l+k-\texttt{length}(\texttt{r1})}, \texttt{r2})) \\
 & & \text{si } k \leq \texttt{length}(\texttt{r1}) \text{ et } \texttt{length}(\texttt{r1}) \leq k+l
\end{array}
$$

Afin de faciliter l'implémentation de ses identités, nous fournissons les fonctions suivantes :

1. `void set_sub_rope(rope r, rope r_sub)` : qui transforme une corde $\texttt{sub}_{i,j}(\texttt{r}')$ en une corde $\texttt{sub}_{i,j}(\texttt{r\_sub})$; le nombre de parents de $\texttt{r}'$ étant décrémenté et celui de `r_sub` étant incrémenté.
2. `void add_to_offset(rope ch, int n)` : qui transforme une corde $\texttt{sub}_{i,j}(\texttt{r}')$ en une corde $\texttt{sub}_{i+\texttt{n},j}(\texttt{r}')$.
3. `void turn_into_append(rope r, rope left, rope right)` : qui transforme la corde `r` de la forme $\texttt{sub}_{i,j}(r')$ en la corde `conc(left, right)`, le nombre de parents de r' étant décrémenté tandis que les nombres de parents de `left` et `right` sont incrémenté.

Exercice 2

Question 2.1 En exploitant les identités précédentes, écrire une fonction `void sub_sub_case(rope r)` qui transforme une corde de la forme $\texttt{r} = \texttt{sub}_{i,j}(\texttt{sub}_{k,l}(\texttt{r}'))$ en une corde plus simple.

Question 2.2 En exploitant les identités précédentes, écrire une fonction `void sub_concat_case(rope r)` qui transforme une corde de la forme $\texttt{r} = \texttt{sub}_{i,j}(\texttt{conc}(\texttt{r1},\texttt{r2}))$ en une corde plus simple.

Question 2.3 A l'aide des fonctions construites dans l'exemple précédent, écrire une fonction `void push_sub_nodes(rope r)`, qui *normalise* la corde `r`, c'est-à-dire qui transforme `r` en une corde telle que pour tout les nœud de la forme $\texttt{sub}_{i,j}(\texttt{r}')$, $\texttt{r}'$ soit de la forme $\texttt{ch}_w$.

Question 2.4 Ecrire une fonction `void destroy(rope r)`(r) qui libère la mémoire occupée par r et ses descendants, si ils n'ont plus de parents.

Question 2.5 On dit qu'une corde r est *superficiellement équilibrée* si sa hauteur (la hauteur de l'arbre qui la représente) est inférieure ou égale à `ln2(length(r))` + 1 (en supposant `length(r)` > 0 et qu'un arbre réduit à une feuille est de hauteur nulle). Une corde r est *équilibrée* si elle est superficiellement équilibrée en chacun de ses nœuds et si $\texttt{sub}_{i,j}(r')$ est un nœud de r alors r' est une feuille. Ecrire une fonction de concaténation `rope balanced_concat(rope r1,rope r2)` qui étant donnée deux cordes équilibrées `r1` et `r2` renvoie une corde équilibrée `r` telle que $\texttt{conc}(\texttt{r1}, \texttt{r2}) \equiv \texttt{r}$. NB : bien faire attention à respecter la propriété de persistance des cordes. Le logarithme en base 2 est calculé par la fonction `int ln2(int n)`.
