Pile avec accès au k-ème élément

L'objectif est d'implémenter une pile où la primitive `stack_top(s)` permettant de lire le sommet de la pile `s` est remplacée par celle-ci :

```
/* return the k-th top element of a stack s containing enough element */
extern void * stack_get(stack s, int k);
```

les autres primitives restant identiques (cf. `stack.h`). Le sommet de la pile est donc accédé en faisant `stack_get(s,0)`. On va faire en sorte d'accéder au k-ème élément en temps $O(\min\{k, \log n\})$ où n est la taille de la pile.

Le principe est d'implémenter la pile comme une liste chaînée d'arbres binaires complets. La tête de liste représente les éléments au sommet de la pile (dernièrement empilés), chaque élément de la pile étant stocké dans un nœud de l'arbre selon un parcours en profondeur d'abord. En particulier le sommet de la pile est stocké dans la racine de l'arbre de la tête de la liste. Les $n/2$ éléments suivants (pour une arbre ayant $n+1$ nœuds) sont stockés à gauche, puis les $n/2$ suivants dans le sous-arbre droit.

Le i-ème élément d'un l'arbre binaire complet est trouvé par une recherche dichotomique de i dans l'arbre qui doit être basé sur le nombre de nœuds de l'arbre. L'entier i représente le rang des nœuds de l'arbre et ne doit en aucun cas être stocké.

Il est nécessaire de stocker au niveau des cellules de la liste (et non des nœuds de l'arbre) la hauteur de l'arbre ou de manière équivalente le nombre de nœuds qu'il contient. On rappelle qu'un arbre binaire complet de hauteur h possède précisément $n = 2^{h+1} - 1$ nœuds. En `C`, on peut calculer facilement 2^i avec l'expression `1<<i`. Notez bien qu'ici la hauteur est le nombre d'arêtes (et non de nœuds) pour atteindre une feuille à partir de la racine.

La principale idée lors de l'empilement est de fusionner deux arbres consécutifs dans la liste dès lors qu'on peut les remplacer par un arbre binaire complet de taille double. En fait cette opération ne peut pas se produire en cascade. Elle ne peut se faire qu'une seule fois entre les deux premiers arbres de la liste s'ils sont de même taille. Cela garantit un empilement en temps constant. Le déplilement, inversement, découpe l'arbre en tête de liste en deux arbres de même taille en supprimant sa racine qui représente le sommet de le pile.

Dans un deuxième temps, il conviendra d'optimiser l'espace mémoire en considérant deux types de nœuds. Les noeuds feuilles et les autres, l'idée étant qu'il n'est pas nécessaire de stocker les fils des feuilles qu'on sait être `NULL`. Malheureusement en `C` un pointeur vers des objects de tailles différentes (type nœud ou feuille) doit se faire via le type `void*` et des *cast*. Au prix d'une gestion plus complexe de la structure de données, cela permet d'économiser près de la moitiés des pointeurs puisque la moitié (moins un) des nœuds d'un arbre binaire complet sont des feuilles.