TP avancé: Arbres 2-3-4

MP2I Lycée Pierre de Fermat

Dans ce TP, on s'intéresse à un type d'arbres proche des arbres binaires de recherche, appelé les arbres 2-3-4. Tout le TP se fera en C.

Définition 1. Un arbre strict est un arbre dans lequel chaque noeud a soit tous ses enfants vides, soit aucun enfant vide.

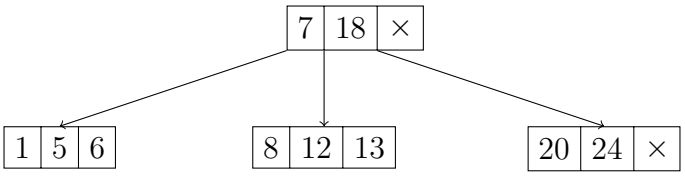
Un **arbre 2-3-4** est un arbre strict constitué de nœuds ayant 2, 3 ou 4 enfants, et tel que :

— Pour $t \in \{2, 3, 4\}$, si un nœud a t enfants $A_1, \ldots, A_t$, alors il est étiqueté par $t-1$ valeurs $v_1, \ldots, v_{t-1}$ de telle sorte que :
  - Toute étiquette e_1 de A_1 vérifie $e_1 \leq v_1$
  - Toute étiquette e_2 de A_2 vérifie $v_1 < e_2 \leq v_2$
  - Toute étiquette e_3 de A_3 vérifie $v_2 < e_3 \leq v_3$
  - ...
  - Toute étiquette e_t de A_t vérifie $v_t < e_t$

— Toutes les feuilles sont à la même profondeur.

On notera $N_2(t_1, x_1, t_2)$ un arbre 2-3-4 dont les deux enfants sont t_1 et t_2, et dont l'unique étiquette est x_1. On utilise de même les constructeurs N_3 (resp. N_4) pour les arbres à 3 (resp. 4) enfants.

Par exemple, l'arbre A_0 suivant est un arbre 2-3-4 valide, de hauteur 1, contenant les étiquettes de l'ensemble $\{1, 5, 6, 7, 8, 12, 13, 18, 20, 24\}$ (les arbres vides enfants des feuilles ne sont pas représentés) :



La racine de cet arbre a 2 étiquettes et 3 enfants. Son enfant le plus à gauche a 3 étiquettes, et 4 enfants vides.

On appelle 2-nœud un nœud ayant deux enfants, 3-nœud un nœud ayant trois enfants, et 4-nœud un nœud ayant quatre enfants. On note un nœud dans l'ordre infixe, en omettant les arbres vides, par exemple l'arbre précédent s'écrira :

$$N_3(N_4(1, 5, 6), 7, N_4(8, 12, 13), 18, N_3(20, 24))$$

Question 1. Dessinez un arbre 2-3-4 de hauteur 2 contenant tous les entiers 1 à 22.

Question 2. Justifiez que la hauteur d'un arbre 2-3-4 est logarithmique en sa taille.

On définit le type suivant en C permettant de représenter les nœuds des arbres 2-3-4 :

```c
typedef struct b_tree {
  int n_children; // vaut 2, 3, 4, ou 5
  struct b_tree* children[5]; // enfants
  int labels[4]; // étiquettes
  struct b_tree* parent;
} b_tree_t;
/* Invariants de la structure:
- les n_children premières cases de children sont utilisées
- les n_children-1 premières cases de labels sont utilisées
- n_children vaut 2, 3 ou 4, mais peut valoir 5 momentanément
  au sein de l'exécution d'une insertion */
```

On choisira arbitrairement que le parent de la racine d'un arbre est `NULL`.

La procédure d'insertion demandera à certains nœuds d'avoir momentanément 4 étiquettes et 5 enfants. Cependant, en dehors de cette procédure, les arbres auront toujours exactement 2, 3, ou 4 enfants.

Question 3. Écrivez une fonction de signature :

`b_tree_t* b_tree_create2(b_tree_t* t1, int x1, b_tree_t* t2)`

créant l'arbre $N_2(t_1, x_1, t_2)$

Question 4. Sur le même modèle, écrire deux fonctions `b_tree_create3` et `b_tree_create4` représentant les constructeurs N_3 et N_4 :

```c
b_tree_t* b_tree_create3(b_tree_t* t1,
                         int x1,
                         b_tree_t* t2,
                         int x2,
                         b_tree_t* t3);

b_tree_t* b_tree_create4(b_tree_t* t1,
                         int x1,
                         b_tree_t* t2,
                         int x2,
                         b_tree_t* t3,
                         int x3,
                         b_tree_t* t4);
```

Question 5. Écrivez une fonction `void test()` dans laquelle, pour l'instant, vous créerez l'arbre dessiné plus haut. Vous viendrez alimenter cette fonction avec des tests à chaque nouvelle fonction écrite.

Question 6. Écrivez une fonction de recherche `bool b_tree_search(b_tree_t* t, int x)` déterminant si x se situe dans t.

Question 7. Écrivez une fonction d'affichage `void b_tree_print(b_tree* t)` affichant un arbre dans l'ordre infixe.

On s'intéresse maintenant à la procédure d'insertion dans les arbres 2-3-4. Pour insérer x, on commence par trouver la seule feuille qui peut contenir x, i.e. le dernier nœud visité lors de la recherche. On rajoute x sur ce nœud, qui peut alors contenir 4 étiquettes. On fait alors remonter l'anomalie vers la racine en **éclatant** les nœuds à partir de cette feuille :

Algorithme 1 : Éclater

Entrée(s) : A un nœud d'un arbre 2-3-4 contenant entre 1 et 4 étiquettes
Sortie(s) : Nouvelle racine de l'arbre contenant A, ou NULL si la racine n'a pas changé. Cet arbre est modifié pour être un arbre 2-3-4 valide, sans changer son contenu.

```
si A contient 3 étiquettes ou moins alors
    retourner NULL
Notons A_1, A_2, A_3, A_4, A_5 les 5 enfants de A, et e_1, e_2, e_3, e_4 ses quatre étiquettes ;
G ← N(A_1, e_1, A_2);
D ← N(A_3, e_3, A_4, e_4, A_5);
si A est la racine alors
    retourner N(G, e_2, D)
sinon
    P ← parent de A. ;
    Rajouter à P l'étiquette e_2, et y remplacer A par G et D;
    retourner Éclater P
```

Algorithme 2 : Insérer

Entrée(s) : A un arbre 2-3-4, x un élément à insérer
Sortie(s) : Nouvelle racine de A. Cet arbre est modifié pour contenir aussi x.

```
C ← A// sous-arbre courant de A
tant que C n'est pas une feuille faire
    Comparer x au étiquettes de C pour trouver D l'enfant de C pouvant contenir x;
    C ← D;
Ajouter x aux étiquettes de C;
// x est bien positionné
R ← Éclater(C);
si R est NULL alors
    retourner A
sinon
    retourner R
```

Question 8. Appliquez l'algorithme d'insertion pour insérer la clé 2 dans l'arbre A_0 décrit plus haut. Dessinez l'état de l'arbre après avoir positionné l'étiquette sur la feuille adéquate, puis l'état de l'arbre après avoir lancé la procédure "Éclater".

Question 9. On note A'_0 l'arbre obtenu après l'insertion précédente. Appliquez l'algorithme d'insertion pour insérer la clé 10 dans l'arbre A'_0. Dessinez l'état de l'arbre après avoir positionné l'étiquette sur la feuille adéquate, puis l'état de l'arbre après avoir lancé la procédure "Éclater".

Lors de l'insertion, une fois la feuille adéquate trouvée, il faut ajouter x à ses étiquettes. Cet ajout doit conserver l'ordre des étiquettes.

Question 10. Écrivez une fonction `void int_tab_insert (int* t, int i, int x, int n)` qui insère x à la case i de t, en décalant les cases $t[i], \ldots, t[n-1]$ d'une place vers la droite. Par exemple, si $t = [A, B, C, ...]$, alors l'appel `int_tab_insert (t, 1, D, 3)` modifie t en $[A, D, B, C, ...]$.

Question 11. Écrire une fonction `void tree_tab_insert (b_tree_t** t, int i, b_tree_t* x, int n)` agissant exactement de la même manière sur les tableaux de pointeurs d'arbres.

Question 12. Écrire une fonction `eclater (b_tree_t* t)` implémentant la procédure **Éclater** décrite plus haut.

Question 13. Implémenter une fonction `b_tree_insert (b_tree_t* t, int x)` insérant x dans t selon l'algorithme décrit plus haut. On aura comme précondition que x n'est pas déjà une étiquette de t.

Les arbres 2-3-4 sont un cas particuliers des ***B-arbres****, dans lesquels chaque noeud peut stocker entre t et $2t$ étiquettes, où t est un paramètre fixe. Pour les arbres 2-3-4, on a donc $t = 2$. L'avantage principal des B-arbres par rapport aux arbres binaires de recherche comme les arbres rouge-noir est qu'une recherche / insertion va suivre moins de pointeurs ($\mathcal{O}(\log_2(n))$ pour les arbres binaires, $\mathcal{O}(\log_t(n))$ pour les B-arbres). Dans un ordinateur, la mémoire est divisée en blocs contigus, de telle sorte qu'accéder à deux cases mémoires d'un même bloc est rapide, mais accéder à deux cases mémoires se trouvant dans des blocs distincts coûte cher. Les B-arbres peuvent tirer parti de ce système, en regroupant les données au maximum au sein des noeuds, minimisant le nombre de sauts entre blocs.*