Arbres d'intervalles

Dans ce TP, on va étudier les arbres d'intervalles, qui sont des arbres binaires de recherche spécialisés dans le stockage d'intervalles.

On veut savoir si un événement **e**, ou une partie de l'événement **e**, s'est déroulé simultanément avec au moins l'un des événements $\mathbf{e}_k$ appartenant à un ensemble $\{\mathbf{e}_1, \ldots, \mathbf{e}_n\}$ d'événements. Quitte à identifier chaque événement **e** avec l'intervalle de temps **I** durant lequel il s'est produit, on se ramène donc à déterminer, étant donné un ensemble $\{\mathbf{I}_1, \ldots, \mathbf{I}_n\}$, si un intervalle **I** a un point commun avec au moins l'un des intervalles $\mathbf{I}_k$. Pour ce faire, on va utiliser des arbres binaires de recherche spécifiques.

Dans toute la suite, on supposera que l'on ne manipule que des intervalles fermés, c'est-à-dire de la forme $\mathbf{I} = [a, b]$, et on dira que deux intervalles *se rencontrent* si leur intersection n'est pas vide. Par ailleurs, si $\mathbf{I} = [a, b]$ et $\mathbf{I}' = [a', b']$ sont deux intervalles, on considérera que **I** est *strictement plus petit que* $\mathbf{I}'$ dans les deux cas suivants : si $a < a'$, et si $a = a'$ et $b < b'$.

Pour représenter des intervalles et des arbres binaires de recherche contenant des intervalles, on utilisera les types suivants :

```c
typedef struct {
  int min;  /* minimum de l'intervalle */
  int max;  /* maximum de l'intervalle */
} Intervalle;

typedef struct noeudI {
  Intervalle i;      /* intervalle représenté */
  int min;           /* entier min contenu dans un intervalle de l'arbre */
  int max;           /* entier max contenu dans un intervalle de l'arbre */
  struct noeudI *fg; /* fils gauche */
  struct noeudI *fd; /* fils droit */
} NoeudI, *ArbreI;
```

0. **Sortir un papier et un crayon** — Si vous ne réussissez pas cet exercice, votre enseignant ne viendra pas vous aider en cas de problème lors d'une des questions qui suivent.
1. **Intersection entre deux intervalles** — Dessiner les différents cas possibles, puis écrire une fonction `int rencontre(Intervalle I, Intervalle J)` qui renvoie `1` si les intervalles `I` et `J` se rencontrent, et `0` sinon.
2. **Ajout dans un ensemble d'intervalles**
   a. Dessiner l'arbre obtenu après avoir ajouté successivement les intervalles $[19, 22]$, $[27, 31]$, $[10, 12]$, $[25, 28]$, $[3, 11]$, $[14, 16]$ et $[12, 17]$.
   b. Écrire une fonction `void ajoute(Intervalle I, ArbreI *a)` qui ajoute l'intervalle `I` dans l'arbre `*a` s'il n'y est pas déjà présent.
   Quelle est sa complexité, en fonction de la hauteur h et du nombre de nœuds n de l'arbre `*a` ?

3. **Intersection avec un ensemble d'intervalles** — On souhaite tester si un intervalle `I` rencontre au moins un intervalle parmi ceux stockés dans l'arbre `a` et, si oui, trouver un intervalle stocké dans `a` qui rencontre `I`. Par la même occasion, on souhaite comprendre pourquoi il est effectivement utile d'avoir introduit les champs `min` et `max` dans le type `Noeud`.

   a. Écrire une fonction `ArbreI rencontreNaive(Intervalle I, ArbreI a)` qui renvoie l'adresse d'un nœud `n` contenu dans l'arbre `a` et dont l'intervalle rencontre `I`, si un tel nœud existe, et qui renvoie `NULL` sinon. Cette fonction ne devra **pas** faire appel aux champs `min` et `max` stockés dans chaque nœud.

   Quelle est sa complexité, en fonction de h et de n ?

   b. On suppose que l'intervalle `I` ne rencontre pas l'intervalle stocké dans la racine de l'arbre `a`, et que cette racine admet un fils gauche. Pourquoi peut-on être sûr que

   — si `(a->fg).max` $<$ `I.min`, alors `I` ne rencontre aucun intervalle du sous-arbre `a->fg` ?

   — si `I` est plus petit que `a->i`, alors `I` ne rencontre aucun intervalle du sous-arbre `a->fd` ?

   — si `a->i` est plus petit que `I` et `I.min` $\leqslant$ `(a->fg).max`, alors `I` rencontre forcément un intervalle (au moins) du sous-arbre `a->fg` ?

   c. Écrire une fonction `ArbreI rencontre(Intervalle I, ArbreI a)` qui renvoie l'adresse d'un nœud `n` contenu dans l'arbre `a` et dont l'intervalle rencontre `I`, si un tel nœud existe, et qui renvoie `NULL` sinon. Cette fonction pourra faire appel au champ `max` stocké dans chaque nœud, et devra être de complexité **linéaire** en h.

   Pourquoi est-ce mieux que la complexité calculée à la question 3a ?

4. **Suppression d'un ensemble d'intervalles** — On souhaite maintenant supprimer un intervalle d'un arbre binaire de recherche.

   a. Dessiner l'arbre obtenu après avoir supprimé successivement les intervalles $[10, 12]$ et $[27, 31]$ de l'arbre obtenu à la question 2a.

   b. Écrire une fonction `ArbreI extrait(Intervalle I, ArbreI *a)` exrait de l'arbre `*a` le nœud `n` qui représente l'intervalle `I`, et qui renvoie l'adresse de `n` ; si un tel nœud n'existe pas, la fonction devra renvoyer `NULL`.

   Quelle est sa complexité, en fonction de h et de n ?