Travaux pratiques de programmation

On s'attend à ce que tous les candidats traitent de manière au moins partielle chacune des parties du sujet :

— programmation en OCaml,
— programmation en C et
— audit de code.

1 Algorithme des nœuds-chapeaux

L'algorithme des nœuds-chapeaux est un algorithme permettant de gérer efficacement un calendrier de réservations. Il a été inventé en 2003 (puis amélioré en 2009) par Martin Rayrole. Son cadre d'utilisation est le suivant : on veut gérer les réservations d'un ensemble de ressources (toutes identiques) sur des intervalles de temps. On peut penser par exemple à la réservation de chambres d'hôtels, de bande passante ou encore de processeurs dans une grappe de serveurs.

L'intérêt de cet algorithme est que la complexité des opérations de base (qui sont la vérification de la disponibilité d'une quantité de ressources sur un intervalle de temps, la réservation et l'annulation d'une réservation) se font en temps logarithmique sur la taille (définie plus loin) de l'intervalle de temps total dans lequel effectuer les réservations. En particulier, la complexité des opérations est indépendante du nombre de réservations ou du nombre de ressources proposées à la réservation.

Cette étude de l'algorithme des nœuds-chapeaux est divisée en deux parties indépendantes. Tout d'abord, nous allons définir en langage OCaml cette structure de données ainsi que les fonctions permettant de la manipuler efficacement.

Ensuite, nous mettrons en place en langage C une petite architecture client/serveur simulant un gestionnaire en ligne de réservations, s'appuyant sur l'utilisation de l'algorithme de nœuds-chapeaux dont l'implémentation en C est fournie sous forme d'une bibliothèque.

1.1 Programmation en OCaml

Étant donnés deux entiers $a \leq b$, on définit l'*intervalle d'entiers* $[\![a, b]\!]$ comme l'ensemble des entiers compris entre a et b, bornes incluses :

$$[\![a, b]\!] = \left\{ n \in \mathbf{N} \;\middle|\; a \leq n \leq b \right\}$$

La *taille* de cet intervalle est son cardinal, égal à $b - a + 1$.

Par exemple, $[\![2, 5]\!]$ représente l'ensemble $\{2, 3, 4, 5\}$, de taille 4.

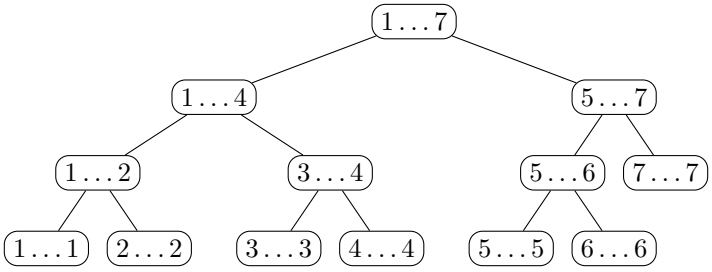
1.1.1 Nœuds-chapeaux dans un arbre calendrier

Un **arbre calendrier** représentant un intervalle d'entiers $[\![a, b]\!]$ (pour $a \leq b$) est défini de la façon suivante :

— la valeur de la racine de cet arbre est l'intervalle `{debut = `a`; fin = `b`}` ;
— si $a = b$ alors ses deux sous-arbres sont vides, sinon ce sont les arbres calendriers représentant $[\![a, m]\!]$ à gauche, et $[\![m + 1, b]\!]$ à droite, où $m = \lfloor \frac{a+b}{2} \rfloor$ est la moyenne arrondie inférieurement de a et de b (ce que l'on peut définir simplement en OCaml en écrivant `let m = (a + b) / 2`).

Par exemple, l'arbre calendrier pour l'intervalle $[\![1, 7]\!]$ est [1] :

1. Les arbres vides ne sont pas représentés.



Les arbres calendrier seront représentés en OCaml sous la forme d'un élément de type `intervalle arbre`, pour les types suivants :

```
type intervalle = { debut : int; fin : int }
type 'a arbre = V | N of 'a arbre * 'a * 'a arbre
```

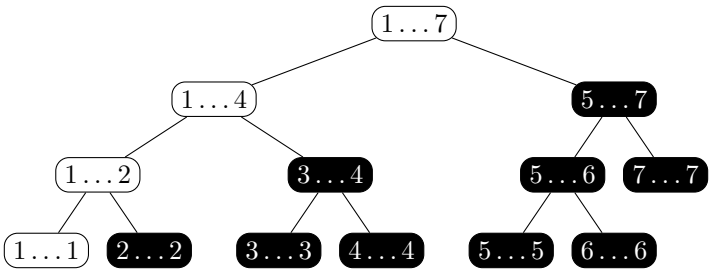
Question 1. Écrire une fonction

`creer_arbre_calendrier : int -> int -> intervalle arbre`

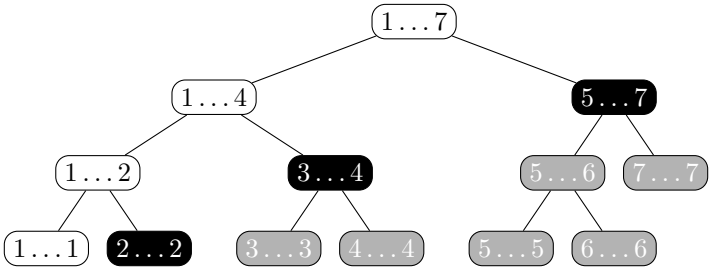
de telle sorte que `creer_arbre_calendrier debut fin` renvoie l'arbre calendrier représentant l'intervalle $[\![$`debut`$,$`fin`$]\!]$. On suppose que l'on a `debut` $\leq$ `fin`, hypothèse que l'on conservera dans la suite de l'énoncé.

Étant donné un arbre calendrier A représentant l'intervalle I et un intervalle $J \subseteq I$, les **nœuds-chapeaux** de J dans A sont les nœuds de l'arbre dont l'intervalle K est inclus dans J et qui sont, parmi ces nœuds, maximaux pour l'inclusion : il n'existe pas dans l'arbre de nœud d'intervalle K' tel que $K \subset K' \subseteq J$ (où $\subset$ désigne l'inclusion stricte).

Considérons par exemple l'intervalle $J = [\![2, 7]\!]$ dans l'arbre calendrier précédent représentant $I = [\![1, 7]\!]$. Les nœuds dont l'intervalle K est inclus dans J sont représentés en noir :



Parmi eux, on ne garde que les éléments maximaux pour l'inclusion, ici au nombre de 3 :



Ainsi, les nœuds-chapeaux de $[\![2, 7]\!]$ dans A sont les nœuds de valeurs $[\![2, 2]\!]$, $[\![3, 4]\!]$ et $[\![5, 7]\!]$.

Question 2. Écrire une fonction

`noeuds_chapeaux : intervalle arbre -> int -> int -> intervalle list`

telle que `noeuds_chapeaux arbre debut fin` renvoie la liste des intervalles des nœuds-chapeaux de l'intervalle $[\![$`debut`$,$`fin`$]\!]$ dans `arbre`. L'ordre des intervalles dans la liste résultat n'est pas spécifié.

L'implémentation devra garantir une complexité *linéaire en la hauteur de l'arbre*. Cette hauteur est, pour un arbre calendrier, logarithmique en la taille de l'intervalle représenté par l'arbre.

1.1.2 Gestion de réservations de ressources

Nous allons maintenant modifier notre approche pour pouvoir tenir compte de réservations. Pour cela, nous allons utiliser des `reservation arbre`, où le type `reservation` est défini ainsi :

```
type reservation = {
  debut : int;
  fin : int;
  capacite : int;
  mutable resa_locale : int;
  mutable resa_globale : int;
}
```

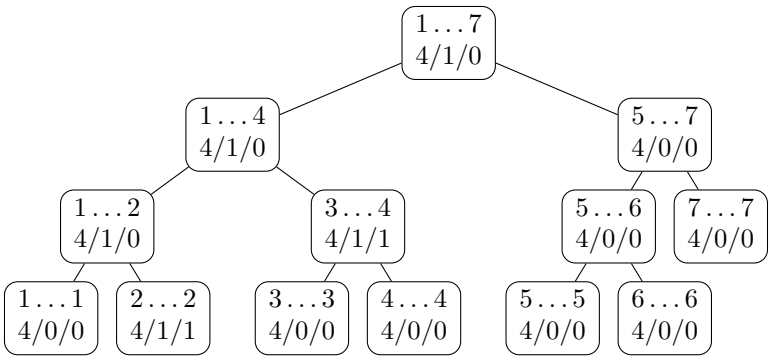
- Les champs `debut` et `fin` indiquent un intervalle de temps.
- La `capacite` est égale à la quantité de ressources proposées à la réservation. Toutes les valeurs de `capacite` des nœuds d'un même arbre sont donc égales, ce champ ayant pour objectif de simplifier l'implémentation.
- Lorsque l'on effectue la réservation d'une quantité q de ressources sur un intervalle J dans un arbre de réservation A (en considérant que celle-ci est possible), on ajoute q à la `resa_locale` de chacun des nœuds-chapeaux de J pour A.
- La valeur de `resa_globale` est définie par l'**invariant structurel** suivant :

  > Pour tout nœud, sa `resa_globale` est égale au maximum des sommes des `resa_locale` le long de tout chemin de ce nœud (inclus) jusqu'à une feuille (incluse).

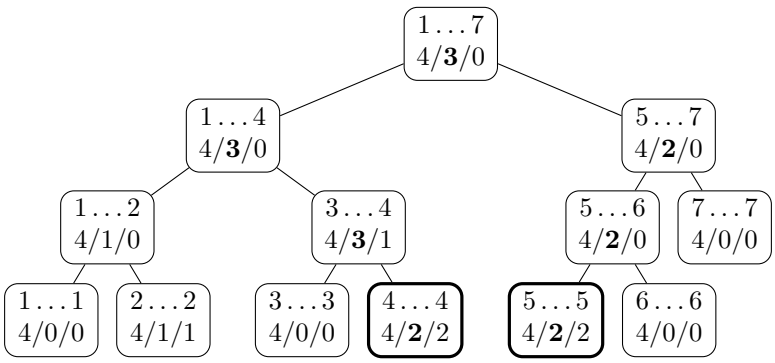
  En particulier, pour tout nœud qui n'est pas une feuille, sa `resa_globale` est égale à la somme de sa `resa_locale` et du maximum des `resa_globale` de ses fils gauche et droit.

Ainsi, lors de la réservation d'une quantité q de ressources sur un intervalle J dans un arbre calendrier A (sous réserve que celle-ci est possible), on ajoute q aux `resa_locale` de tous les nœuds-chapeaux de J pour A, en effectuant les mises à jour nécessaires des `resa_globale`.

Par exemple, voici un arbre de réservation de capacité 4, où l'on a effectué une première réservation d'une ressources sur l'intervalle $[\![2, 4]\!]$. Pour chaque nœud, on indique sur la seconde ligne les valeurs `capacite/resa_globale/resa_locale`.



Si l'on réserve ensuite deux ressources sur l'intervalle $[\![4, 5]\!]$, l'arbre devient (on indique en gras les nœuds-chapeaux ainsi que les modifications de réservations globales) :



Question 3. Écrire une fonction

```
creer_arbre_reservation : int -> int -> int -> reservation arbre
```

telle que `creer_arbre_reservation debut fin capa` renvoie l'arbre de réservation pour l'intervalle $[\![$`debut`, `fin`$]\!]$ de capacité initiale `capa` et où tous les nœuds ont des réservations locales nulles.

Question 4. Écrire une fonction

```
effectuer_reservation : reservation arbre -> int -> int -> int -> unit
```

telle que `effectuer_reservation calendrier debut fin quantite` modifie l'arbre passé en argument pour tenir compte de la réservation de `quantite` ressources sur l'intervalle $[\![$`debut`, `fin`$]\!]$. On suppose que cette réservation est possible ; on ne demande donc pas de le vérifier.

Dans un arbre de réservation A, la réservation d'une quantite q de ressources sur l'intervalle J est possible si et seulement si pour tout nœud-chapeau K de J pour A, la somme de q, de la `resa_globale` de K et des `resa_locale` de tous les nœuds situés sur le chemin entre K (exclu) et la racine de A (incluse) est inférieure ou égale à la capacité de l'arbre.

Question 5. Écrire une fonction

```
reservation_possible : reservation arbre -> int -> int -> int -> bool
```

telle que `reservation_possible arbre debut fin quantite` indique s'il est possible de réserver `quantite` ressources sur l'intervalle $[\![$`debut`, `fin`$]\!]$.

Pour finir, nous allons nous intéresser à l'annulation d'une réservation. La suppression de la réservation de q ressources sur un intervalle J dans un arbre de réservations A est possible si et seulement si la réservation locale de tout nœud-chapeau de J pour A est supérieure ou égale à q, et il suffit alors de diminuer la réservation locale d'autant pour annuler la réservation.

Question 6. Écrire une fonction

```
annulation_possible : reservation arbre -> int -> int -> int -> bool
```

telle que `annulation_possible arbre debut fin quantite` indique s'il est possible d'annuler la réservation de `quantite` ressources sur l'intervalle $[\![$`debut`, `fin`$]\!]$ dans `arbre`.

Question 7. Écrire une fonction

```
annuler_reservation : reservation arbre -> int -> int -> int -> unit
```

telle que `annuler_reservation arbre debut fin quantite` effectue l'annulation de la réservation de `quantite` ressources sur l'intervalle $[\![$`debut`, `fin`$]\!]$ dans `arbre`, en supposant qu'une telle annulation est possible.

1.2 Programmation en C

Le contexte de cette partie est la programmation d'une application qui propose à ses utilisateurs de partager un calendrier en réseau. Ce calendrier partagé est implanté selon l'algorithme des noeuds-chapeaux. Il est donc possible pour un utilisateur, comme spécifié dans `calendrier.h` et implanté dans la bibliothèque `libcalendrier.so` fournie, de : créer un calendrier, vérifier si une réservation est possible, faire une réservation ou annuler une réservation.

Chaque utilisateur de l'application possède une vue locale du calendrier $\mathcal{C}_\ell$ qu'il met à jour en envoyant des requêtes à une application serveur qui possède le calendrier principal $\mathcal{C}_\mathcal{P}$. Un calendrier est partagé par un ensemble d'utilisateurs que l'on appellera les `peers`. Dans une première version de cette application, on ne cherche pas à obtenir une copie complète du calendrier principal dans la vue locale $\mathcal{C}_\ell$. Les entrées dans le calendrier local sont uniquement les entrées valides de l'utilisateur ℓ dans le calendrier principal.

L'objectif principal de cet exercice est la programmation de l'envoi d'une requête de l'application cliente d'un `peer` vers l'application serveur via un *socket TCP/IP*. Le première partie définit le protocole de communication et vous demande de l'implanter dans le module `protocole` fourni. La seconde vous demande d'écrire les sous-programmes qui permettent d'envoyer la requête et de recevoir la réponse associée dans les programmes principaux `cal_client.c` et `cal_serveur.c`. La dernière partie s'intéresse à étendre l'application pour passer d'un mode de communication client-serveur à un mode de communication pair-à-pair.

Note : Un ensemble de ressources est mis à votre disposition :
- les fichiers source et bibliothèques à compléter ou utiliser pour répondre aux questions suivantes ;
- un fichier `Makefile` qui permet de compiler l'ensemble des programmes avec les éventuelles bibliothèques nécessaires. Ce `Makefile` utilise le compilateur `gcc` et non `gcc -c99` délibérément ;
- une note qui rappelle les bases pour programmer une communication via un socket TCP/IP en C.

Dans l'environnement `Nonos`, il faudra inclure le chemin vers le répertoire courant dans la variable d'environnement `LD_LIBRARY_PATH` pour que l'environnement d'exécution trouve les bibliothèques fournies. On peut par exemple exécuter la commande suivante si votre répertoire de travail est `/home/candidat/tp` :

```
export LD_LIBRARY_PATH=/home/candidat/tp:$LD_LIBRARY_PATH
```

1.2.1 Protocole de communication

Le client est en mesure de vérifier si une réservation est possible, de faire une réservation ou d'annuler une réservation le concernant. Le calendrier ne gère pas le fait qu'une réservation soit associée à un utilisateur donné. À chaque requête correspond une réponse qui indique si la demande est réalisée avec succès ou non.

La requête est un message qui comporte les 4 champs suivants :

Champ	Taille (octet)	Type C
`TYPE`	2	`uint16_t`
`DEBUT`	2	`uint16_t`
`FIN`	2	`uint16_t`
`QUANT`	2	`uint16_t`

et la réponse un message qui comporte les 2 champs suivants :

Champ	Taille (octet)	Type C
`TYPE`	2	`uint16_t`
`RESP`	2	`uint16_t`

avec `RESP == 1` si la requête est effectuée avec succès, 0 sinon. Le champ `TYPE` des deux messages permet de distinguer la nature du message en codant par une valeur entière le fait qu'il soit : une requête (resp. réponse) de mise à jour ; une requête (resp. réponse) de vérification de disponibilité ; une requête (resp. réponse) d'annulation.

Question 8. Implanter le protocole de communication décrit précédemment de façon à pouvoir réaliser les opérations de réservation de période, de vérification de disponibilité sur une période et de suppression

d'une réservation. Compléter à cet effet les fichiers `protocole.h` et `protocole.c` fournis et le fichier de test `test_protocole.c`. Il est fortement conseillé de définir des structures de données de type enregistrement pour les types `msg_request_t` et `msg_response_t`.

1.2.2 Programmer l'échange requête — réponse en mode client / serveur

Pour déployer le protocole applicatif précédent, on s'appuie sur un empilement protocolaire standard TCP/IPv4. Chaque échange de message donnera lieu à la création d'un socket de communication dédié à l'échange uniquement. Les prochaines questions s'intéressent d'abord à la connexion d'un socket entre un client et un serveur de sockets, puis à l'échange de deux messages de type requête-réponse. Une note `programmer_socket.pdf` est fournie pour rappeler les principales étapes de la programmation d'un socket.

Question 9. À l'aide du canevas `cal_serveur.c` founi, renseigner les principales étapes qui permettent :
- de créer le serveur de socket ;
- de créer l'adresse du serveur de socket `serv_addr` à partir du port TCP de l'application défini dans `protocole.h` et d'une adresse IPv4. Cette dernière est positionnée à `INADDR_ANY`. Dans la troisième partie, elle sera donnée en argument de la ligne de commande pour tester votre application en mode pair-à-pair.
- d'associer le serveur de socket et l'adresse du serveur de socket `serv_addr` ;
- d'activer le serveur de socket (*i.e.* le mettre en écoute de demandes de connexion entrantes) ;
- d'accepter une demande de connexion et ainsi générer le socket utilisé pour l'échange de messages.

Les principaux sous-programmes à modifier sont `creer_serveur_socket` et `main`. On s'attachera à avertir l'utilisateur des éventuels échecs liés à la mise en place de la connexion (vous pourrez exploiter les sous-programmes définis dans `signale_erreur_socket.c`).

Une bibliothèque `libpeers.so` vous est fournie, ainsi que sa spécification `peers.h`. Elle permet de générer une liste chaînée d'utilisateurs qui partagent un même calendrier. Chaque utilisateur est caractérisé par un identifiant entier unique et une adresse IPv4 encodée sour la forme d'une chaîne de caractères avec une notation décimale à point. Dans l'implantation qui vous est fournie, cette bibliothèque permet de modéliser un utilisateur qui appartient à une famille qui partage le calendrier dans le programme principal de `cal_client.c`.

Question 10. À l'aide du canevas `cal_client.c` fourni, renseigner les principales étapes qui permettent de créer un socket de communication en se connectant au serveur de socket ayant pour adresse IP celle définie pour un utilisateur de la liste de `peers`. Il faudra pour cela modifier les sous-programmes `creer_socket` et `connecter_socks`. On s'attachera à avertir l'utilisateur des éventuels échecs liés à la mise en place de la connexion.

Question 11. Il s'agit maintenant de programmer l'envoi de la requête et l'envoi de la réponse suite à la réception de la requête par le serveur. Compléter les sous-programmes `envoyer_recevoir(...)`, `envoyer_requete(...)` dans `cal_client.c` et `gere_msg(...)` dans `cal_serveur.c` de façon à échanger une requête et une réponse qui permettent de modifier le calendrier partagé. On s'attachera à avertir l'utilisateur des éventuels échecs liés à l'échange de messages.

Question 12. L'implantation du client dans le programme principal de `cal_client.c` réalise l'envoi d'une requête de mise à jour du calendrier. Dans cette question, il vous est demandé de créer des sous-programmes de test des différentes requêtes possibles et de les utiliser dans le programme principal de `cal_client.c`.

1.2.3 Déployer l'application en mode pair-à-pair

On s'attache dans cette partie à passer à une implantation de l'application vers un mode de communication pair-à-pair. Dans ce mode, quand un utilisateur modifie son calendrier localement, la requête de modification est aussi envoyée aux autres pairs pour qu'il mettent à jour leur vue locale [2]. Il n'y a donc

2. Dans un mode de communication client-serveur traditionnel, une application serveur comporterait le calendrier qui fait autorité. Une demande de modification est adressée par un client uniquement à l'application serveur.

pas de calendrier principal. Si aucun message n'est perdu, tous les utilisateurs (*i.e.* les peers) possèdent une réplique du même calendrier. On supposera dans la suite que les applications distribuées sont en mesure d'établir une connexion TCP/IP à tout moment entre elles, et que les échanges de message se font tous avec succès.

Dans cette dernière partie, il est demandé de modifier l'application précédente pour qu'elle puisse être déployée dans un mode de communication pair-à-pair. Pour ce faire, on peut partir des fichiers `cal_client.c` et `cal_serveur.c` que l'on renommera `calpeer_client.c` et `calpeer_serveur.c`. Dans ce mode, la même requête est envoyée par l'utilisateur à tous les utilisateurs du calendrier partagé. L'utilisateur ne met à jour son calendrier qu'une fois une réponse positive reçue de la part des autres pairs.

Question 13. Modifier l'application de façon à obtenir le comportement décrit ci-avant.

Pour tester votre application, il faut générer plusieurs pairs virtuels (*i.e.* plusieurs instances de `calpeer_serveur`), qui écoutent sur différentes adresses IP. Il faut pour cela modifier votre programme de création de serveur de socket pour que l'adresse IPv4 soit lue en argument de la ligne de commande selon une notation décimale à point. Pour émuler plusieurs pairs, on utilisera des adresses IP du réseau `127.0.0.0/8` associé à l'interface de bouclage.

2 Revue de code

2.1 Code en OCaml

L'objectif est d'écrire une fonction qui prend en entrée une liste de type `couleur list` où l'on a défini le type `type couleur = N | B` et renvoie la liste d'entiers indiquant les longueurs des blocs successifs d'éléments `N`.

On veut par exemple :

```
# blocs [B; N; N; B; B; B; N; N; N; N; B; N] ;;
- : int list = [2; 4; 1]
```

Voici une proposition d'écriture de cette fonction :

```
type couleur = N | B

let rec blocs l =
  let l = ref l and r = ref [] and cnt = ref 0 in
  while !l <> [] do
    let (hd :: tl) = !l in
    if hd = B then
      if !cnt > 0 then (
        r := !cnt :: !r;
        cnt := 0)
      else if hd = N then incr cnt;
    l := tl
  done;
  !r
```

Le code est disponible dans le fichier `Revue/revue.ml`.

2.2 Code en SQL

On considère une base de données qui contient 3 tables :

Villes CP (clef primaire), nom, pays (clef étrangère sur le code de pays), habitants

Pays code (clef primaire), nom, continent (clef étrangère sur le code de continent)

Continents code (clef primaire), nom

Un exemple de base de données ayant ce schéma se trouve dans le fichier `Revue/table.db` que l'on peut interroger à l'aide de SQLite en exécutant `sqlite3 Revue/table.db`. On rappelle que dans l'interface en ligne de commandes de SQLite, les requêtes SQL nécessitent un point-virgule final pour être exécutées, et que l'on peut avoir au schéma de la base en exécutant `.schema` (et la liste des commandes spéciales est, comme indiquée, obtenue en exécutant `.help`).

On propose pour audit les requêtes suivantes, que vous pouvez retrouver dans le fichier `Revue/revue.sql` :

```
select cp, sum(habitants) from villes and pays where pays=code group by code
```

```
select nom, continent
from villes, join pays on pays=code, join continent on
continent=code having habitants is not null
```