Ensembles naifs

Dans ce TP on représente des ensembles d'entiers en utilisant plusieurs structures de données différentes et d'expérimenter ces solutions sur une application concrète.

1 Type abstrait ensemble

Le but de l'exercice est d'implémenter de deux manières différentes un type abstrait *ensemble* permettant de représenter des ensembles finis d'éléments comparables entre eux (dans le cas présent ce seront des `long int`). Nous utiliserons pour cela des techniques de programmation vues au S3 : les *listes simplement chaînées* et les *tableaux dynamiques* (triés ou non).

Chaque implémentation fournie devra obligatoirement définir un type `Ensemble` (pointeur vers une structure C) et définir au moins les fonctions suivantes :

- `Ensemble* cree_ensemble()`
- `int taille(const Ensemble *ens)`
- `bool appartient(const Ensemble *ens, long int elem)`
- `int ajoute(Ensemble *ens, long int elem)`

On rappelle que dans un ensemble, chaque entier ne peut apparaître qu'une et une seule fois. Si l'on ajoute plusieurs fois le même élément à un ensemble la taille totale de l'ensemble ne doit donc pas changer.

Ces en-têtes de fonctions sont fournies dans le fichier `ensemble.h` fourni (voir plus bas).

On pourrait imaginer ajouter plusieurs autres fonctions dans cette liste, mais ce n'est pas indispensable pour l'application qui va nous occuper. On évoque néanmoins quelques ajouts possibles au paragraphe 7.1.

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.

2 Application

On propose, afin de mettre à l'épreuve les ensembles ainsi implémentés, une application concrète.

Le site web pour lequel vous travaillez aimerait connaître le nombre de visiteurs uniques en se basant sur leur adresse IP. On vous fournit une liste (potentiellement énorme) d'apdresses IP ayant visité le site, sous la forme d'un fichier au format texte brut `visites.log`. Chaque ligne de ce fichier est constitué d'un *timestamp* (un entier long indiquant une date et une heure) et une adresse IPv4, séparés par un espace. Chaque adresse IPv4 est elle-même constituée de quatre entiers sur un octet (compris entre 0 à 255) séparés par un point. Voici un exemple de début de fichier :

```
1642582836 192.168.0.1
1642582853 192.168.0.2
1642583455 192.168.0.1
1642583456 192.168.0.3
1642583465 192.168.0.4
1642583473 192.168.0.2
1642583480 192.168.0.4
etc.
```

La question posée est de déterminer, le plus efficacement possible, combien d'adresses IP différentes sont contenues dans le fichier `visites.log`. On fournit le code suivant (fichier `visites_uniques.c` téléchargeable sur elearning) pour résoudre ce problème :

```c
#include /* ... */
#include "ensemble.h"

/* convertit une chaîne représentant une adresse IPv4 en entier long */
long int encode_ip(char *ip) {/* ... */}

/* on utilise un ensemble sans connaître son implémentation ! */
int main(int argc, char *argv[]) {
  /* ... */
  FILE *df = fopen(argv[1]);
  // int ndiff=0;
  Ensemble *a = cree_ensemble();
  char ip[16];
  while (fscanf(df, "%*d␣%s", ip) == 1) {
    long int n = encode_ip(ip);
    // ajoute retourne 1 si il y a ajout
    // if (ajoute(a,n)) ndiff+=1;  assert(ndiff == taille(a));
    ajoute(a, n);
  }
  printf("trouvé␣%d␣ip␣différentes\n", taille(a));
  detruit_ensemble(a);
  fclose(df);
}
```

Il reste à concevoir une ou plusieurs implémentations possibles pour le type `Ensemble` et ses fonctions, ce sera l'objet de la suite du TP.

3 Implémentation à l'aide de tableaux (non triés)

On fournit une première implémentation du type `Ensemble` à l'aide de tableaux non triés (fichier `ensemble_tab_non_trie.c`). Cette première partie du TP consiste simplement à prendre connaissance du code fourni, à le compiler et à l'exécuter.

1. Télécharger sur elearning le fichier `tp1-ensembles-naifs.zip` et le décompresser.
2. Ouvrir dans un IDE et lire intégralement les fichiers `ensemble.h`, `ensemble_tab_non_trie.c`, `visites_uniques.c` et `visites_court.log`.
3. Indiquer la complexité asymptotique des fonctions `appartient`, `ajoute` et `taille` en fonction du nombre d'éléments présents dans l'ensemble.
4. Compiler la première version du programme de comptage de visiteurs à l'aide de la commande `make tab_non_trie`, puis exécuter le programme `visites_uniques_tab_non_trie` sur le fichier `visites_court.log`. Vérifier que le résultat obtenu est correct.
5. Tester à nouveau le programme sur le fichier `visites_long.log`. Qu'observe-t-on ?

4 Implémentation à l'aide de listes chaînées

Dans un deuxième temps on représentera un ensemble par une structure possédant deux attributs : un pointeur vers la première cellule d'une liste simplement chaînée (non triée mais sans doublons) et un entier représentant la taille de l'ensemble. On reprend la méthode vue au S3 pour définir les cellules d'une liste chaînée :

```
typedef struct cellule {
  long int valeur          /* entier stocké dans la cellule */
  struct cellule *suivant; /* adresse de la prochaine cellule */
} Cellule;
```

On peut ensuite définir le type `Ensemble` ainsi :

```
typedef struct ensemble {
    Cellule *premier; /* pointeur vers la première cellule ou NULL */
    int taille;       /* nombre d'éléments dans l'ensemble */
} Ensemble;
```

Un ensemble vide est simplement représentée par un attribut `premier` égal à `NULL` et une `taille` égale à 0. On peut donc implémenter la fonction `cree_ensemble` de la manière suivante :

```
Ensemble* cree_ensemble() {
    Ensemble *tmp = malloc(sizeof(Ensemble));
    tmp->premier = NULL;
    tmp->taille = 0;
    return tmp;
}
```

Pour gagner du temps, on fournit également le code des fonctions `cree_cellule`, `detruit_cellule` et `detruit_ensemble`.

1. Ouvrir le fichier `ensemble_lst.c` dans un IDE, et lire attentivement le code fourni.
2. Écrire une fonction *récursive* `Cellule* trouve_cellule(Cellule *cellule, long int elem)` recevant l'adresse de la première cellule d'une liste chaînée, et renvoyant l'adresse d'une cellule de la liste contenant la valeur `elem`, ou `NULL` si une telle cellule n'existe pas.
   Remarque : dans la suite du cours, on utilisera une fonction de ce genre pour rechercher une valeur dans un arbre. Il sera parfois nécessaire d'ajouter un niveau d'indirection et de manipuler des pointeurs de pointeurs ! La signature de la fonction deviendrait donc : `Cellule** trouve_cellule(Cellule ** cellule, long int elem)`.
3. À l'aide de la fonction précédente, compléter la fonction `bool appartient(const Ensemble *ens, long int element)`, qui renvoie le booléen `true` si `element` apparaît dans `ens` et `false` sinon (les constantes `true` et `false` sont définies dans la bibliothèque standard `stdbool.h`).
4. Compléter la fonction `int ajoute(Ensemble *ens, long int elem)` qui ajoute l'élément `elem` à `ens` s'il n'y est pas déjà. La fonction doit aussi maintenir à jour l'attribut `taille` de la structure `ens`.
   Il est recommandé d'utiliser la fonction `appartient` définie précédemment.
   La fonction renverra 1 si l'élément est correctement ajouté, et 0 si déjà présent et -1 cas d'erreur.
5. Compléter la fonction `int taille(Ensemble *ens)` qui renvoie la taille de l'ensemble `ens`. Si le reste du code est bien conçu, cette fonction ne doit pas avoir besoin de parcourir la liste des éléments.
6. Indiquer la complexité asymptotique des fonctions `appartient`, `ajoute` et `taille` en fonction du nombre d'éléments présents dans l'ensemble.
7. Compiler la nouvelle version du programme de comptage de visiteurs à l'aide de la commande `make lst` , puis exécuter le programme `visites_uniques_lst` sur le fichier `visites_court.log`. Vérifier que le résultat obtenu est correct.
8. Tester à nouveau le programme sur le fichier `visites_long.log`. Qu'observe-t-on ?

5 Implémentation à l'aide de tableaux triés

Dans cette dernière implémentation naïve d'ensembles d'entiers, on utilise des tableaux dynamiques comme dans la section 3 mais en choisissant de toujours stocker les éléments par ordre strictement croissant. Cela permettra en particulier de rechercher plus efficacement un élément dans l'ensemble, grâce à l'algorithme de recherche par dichotomie.

1. Créer un fichier `ensemble_tab_trie.c`. Y recopier le contenu du fichier `ensemble_tab_non_trie.c`, on ne touchera qu'aux fonctions `localise`, `appartient` et `ajoute`.
2. Écrire la fonction `bool appartient(const Ensemble *ens, long int elem)`, qui renvoie le booléen `true` si `elem` apparaît dans `ens` et `false` sinon.
   Comme le tableau est maintenant trié, cette fonction peut (et *doit*) faire appel à l'algorithme de dichotomie. On pourra isoler cette recherche dans la fonction `localise`.
3. Écrire la fonction `int ajoute(Ensemble *ens, long int elem)` qui ajoute l'élément `elem` à `ens` s'il n'y est pas déjà. La fonction doit aussi maintenir à jour l'attribut `taille` de la structure `ens`.
   Attention, le nouvel élément doit être ajouté au bon endroit dans le tableau, à la manière du tri par insertion.
   On pourra utiliser la fonction `memmove` pour décaler les éléments du tableau. Il est alors judicieux d'utiliser la fonction `localise` plutôt que `appartient`, **pourquoi ?**
   La fonction renverra 1 si l'élément est correctement ajouté, et 0 en cas ou il est présent et -1 en cas d'erreur (allocation par exemple).
4. Indiquer la complexité asymptotique des fonctions `appartient` et `ajoute` en fonction du nombre d'éléments présents dans l'ensemble.
5. Compiler la nouvelle version du programme de comptage de visiteurs à l'aide de la commande `make tab_trie`, puis exécuter le programme `visites_uniques_tab_trie` sur le fichier `visites_court.log`. Vérifier que le résultat obtenu est correct.
6. Tester à nouveau le programme sur le fichier `visites_long.log`. Qu'observe-t-on ?

6 Bilan

On fait ici un bilan des résultats précédents, et on mesure expérimentalement le temps d'exécution des trois implémentations sur un fichier d'une taille conséquente.

1. Dresser un tableau représentant les complexités des fonctions `appartient`, `ajoute` et `taille` dans les trois implémentations naïves étudiées dans ce TP.
2. À l'aide de l'utilitaire Unix `time`, comparer les temps d'exécutions des trois programmes obtenus sur le fichier `visites_long.log`.

7 Pour aller plus loin (optionnel)

Cette partie du TP est optionnelle, elle sert uniquement à indiquer des prolongements possibles dans l'élaboration de notre type abstrait `Ensemble` et de ses implémentations.

7.1 Fonctions supplémentaires

1. Ajouter au type abstrait `Ensemble` une en-tête de fonction `int supprime(Ensemble *ensemble, long int element)`, écrire cette fonction dans chacune des implémentations réalisées et indiquer sa complexité asymptotique dans chaque implémentation.
   Remarque : Dans le cas de la liste chainée, il sera pratique de disposer de la fonction avec double indirection `Cellule** trouve_cellule(Cellule **cellule, long int elem)` *dont on utilisera une variation pour les arbres.*
2. Ajouter au type abstrait `Ensemble` des en-têtes de fonctions `union`, `intersection` et `difference` permettant de simuler les opérations ensemblistes, les écrire dans chacune des implémentations réalisées et indiquer leur complexité asymptotique dans chaque implémentation.
3. Ajouter au type abstrait `Ensemble` et implémenter les fonctions `min` et `max` permettant de calculer le plus petit et le plus grand élément d'un ensemble et indiquer leur complexité asymptotique dans chaque implémentation.
4. Ajouter au type abstrait `Ensemble` et implémenter les fonctions `extrait_min` et `extrait_max` permettant de calculer le plus petit et le plus grand élément d'un ensemble et de l'en supprimer et indiquer leur complexité asymptotique dans chaque implémentation.
5. Ajouter au type abstrait `Ensemble` et implémenter les fonctions `plafond` et `plancher` permettant de calculer l'élément le plus proche d'un élément donné dans ensemble, respectivement par excès et par défaut, et indiquer leur complexité asymptotique dans chaque implémentation.
6. On appelle *rang* d'un élément e dans un ensemble E contenant des valeurs comparables entre elles le nombre d'éléments strictement inférieurs à e dans E. Par exemple le plus petit élément de E est de rang 0, le second plus petit est de rang 1, etc.
   Ajouter au type abstrait `Ensemble` et implémenter les fonctions `int rang(const Ensemble *ens, long int elem)` et `long int selection(const Ensemble *ens, int rang)` permettant de calculer respectivement le rang d'un l'élément et l'élément d'un rang donné dans l'ensemble, et indiquer leur complexité asymptotique dans chaque implémentation.
   Quand c'est possible, simplifier les autres fonctions en utilisant à bon escient les fonctions `rang` et `selection`.

7.2 Implémentation de tables de symboles

Dans l'application envisagée, on dispose dans le fichier log du serveur de dates de visites (sous la forme de *timestamps*) en plus de l'adresse IP de la machine visitant le site.

On souhaite maintenant connaître non seulement l'ensemble des visiteurs uniques, mais pour chacun d'entre eux la date et l'heure de leur visite la plus récente. Nous devons donc être capable d'*associer* à chaque élément d'un ensemble une valeur quelconque.

1. Modifier le code du TP afin de pouvoir associer à chaque élément d'un ensemble (qu'on appellera dorénavant une *clé*) un entier long quelconque (qu'on appellera une *valeur*), à la manière d'un dictionnaire Python par exemple.
   On ajoutera en particulier un paramètre `valeur` à la fonction `ajoute(Ensemble *ens, long int cle, long int valeur)`, et une nouvelle fonction `long int valeur(Ensemble *ensemble, long int cle)` permettant de récupérer la valeur associée à une clé, si elle existe.