FACULTÉ DES SCIENCES

Formulaire pour reprographie d'examen

Le questionnaire d'examen doit parvenir au Centre de reprographie 5 jours ouvrables avant la date de l'examen.

INFORMATIONS POUR LE SECRÉTARIAT DE LA FACULTÉ
(Attacher à chaque questionnaire)

☒ **INTRA**
☐ **FINAL**

Sigle : IFT 339

Titre : Structures de données

Professeur : Jean Goulet

Date et heure de l'examen : Vendredi, 27 février, 10 h 30

Nombre de pages : 8

Nombre d'étudiants : 502

☐ **Je désire prendre possession des questionnaires d'examen la veille de l'activité et je me rends responsable de leur mise en sécurité.**

Signature : _____

L'étudiant devra répondre

☐ dans un cahier d'examen
☒ sur le questionnaire (en cahier ! SVP)

Veuillez inclure

0 feuilles blanches additionnelles
0 feuilles de papier graphique

Je consens à ce que deux (2) copies du questionnaire de cet examen soient remises à l'AGES (Association générale des étudiants en sciences) après la fin de la période des examens.

☒ **OUI** ☐ **NON**

Signature : _____

Département d'informatique
Examen intratrimestriel
Hiver 2015

IFT339

Structures de données

vendredi 27 février 2015
10 h 30 à 12 h 30
Jean Goulet

Cet examen comprend quatre questions, toutes d'égale valeur. Lisez d'abord tout l'examen, puis répondez directement sur le questionnaire. Il y a en général beaucoup plus d'espace que nécessaire pour vos réponses. L'examen sera corrigé sur 100, puis ramené à 30. Donnez le code demandé en C++. Essayez d'écrire le plus lisiblement possible. Quand on code avec des pointeurs, c'est toujours une bonne idée de commenter le code suffisamment pour que le correcteur comprenne ce que vous vouliez faire... et il faut utiliser avec parcimonie les doubles déréférences de pointeurs!

Comme il a été spécifié au début du cours, vous n'avez droit qu'à la documentation technique, soit le **Résumé C++** jaune. Du papier blanc est disponible pour vos brouillons.

ATTENTION de bien distinguer les questions où on vous demande d'UTILISER un type abstrait de celles où on vous demande d'en IMPLANTER un.

Identifiez-vous lisiblement ci-dessous :

Nom :	
Prénom :	
Matricule :	
Signature :	

1		/25
2		/25
3		/25
4		/25
total		/100

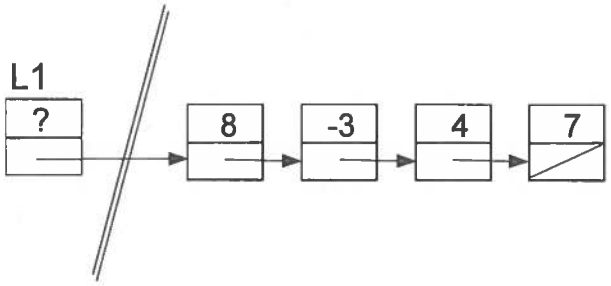
Question 1 – L'implantation de la forward_list de la SL (25 points)

On a implanté la `forward_list` de la SL, une liste unidirectionnelle minimaliste qui ne possède même pas d'attribut donnant le nombre d'éléments de la liste! Cela n'empêche pas cependant d'avoir une fonction `size()` qui retourne le nombre d'éléments de la liste en temps O(n). La représentation classique est une chaine de cellules, avec une cellule supplémentaire d'en-tête pour avoir une position avant le début de la liste. On a ci-dessous une partie de la définition, soit celle que l'on a utilisée pour le troisième laboratoire. Dans la bibliothèque normalisée (SL) il existe aussi la fonction `resize`, qui permet de redimensionner une `forward_list`. Cette fonction permet d'augmenter ou de réduire la dimension. Si on l'augmente, on peut donner une valeur particulière aux éléments ajoutés. Tout ajout ou élimination se fait à la fin de la liste. Par exemple, si on a la liste `L1:<8,-3,4,7>` , l'appel `L1.resize(6,-2)` transforme L1 en `<8,-3,4,7,-2,-2>`. L'appel `L1.resize(2)` la transforme en `<8,-3>`. et l'appel `L1.resize(3,-2)en <8,-3,4>.` Il ne s'agit pas de créer une nouvelle liste mais d'ajouter ou enlever à la liste existante. On voit que la valeur par défaut pour les ajouts est l'objet construit par le constructeur sans paramètres du type des éléments de la liste.

```
template <typename TYPE>
class forward_list{
private:
  struct cellule{
    TYPE CONTENU;
    cellule* SUIVANT;
    cellule(const TYPE& C,cellule* S=nullptr)
       :CONTENU(C),SUIVANT(S){}
  };
  cellule AVANT_DEBUT;

public:

  ...
  size_t size()const;
  void resize(size_t,TYPE=TYPE());
   ...
};
```



Codez les fonctions size et resize de la forward_list.

```
template <typename TYPE>
size_t forward_list::size()const{
```

```
template <typename TYPE>
void forward_list::resize(size_t N, TYPE VAL){
```

Question 2– L'implantation du deque de la SL (25 points)

On a implanté dans le deuxième laboratoire l'équivalent du `deque` de la SL, un vecteur dans lequel l'ajout et l'élimination d'éléments au début ou à la fin est O(1) amorti. On a déjà codé le `push_front` (ajout au début) et le `push_back` (ajout à la fin). On a oublié par contre trois fonctions importantes, que l'on vous demande de coder ici : `pop_front`, qui enlève l'élément du début, `pop_back` qui enlève l'élément de la fin (toutes les deux O(1)) et `erase` qui enlève un élément quelconque (malheureusement O(n) dans le pire cas). Les deux premières sont très simples à coder. La troisième présente une petite difficulté : elle reçoit en paramètre un itérateur vers l'élément à enlever. Il faut alors enlever cet élément logiquement alors qu'on ne peut pas l'enlever physiquement. Il faut décaler les autres éléments pour prendre sa place. Si cet élément est plus proche du début que de la fin, on a avantage à décaler les éléments du début jusqu'à celui-là. S'il est plus proche de la fin, on est mieux de décaler à partir de la fin jusqu'à cet élément. Dans le pire cas, on devra donc recopier n/2 pointeurs. Notez que puisqu'on utilise des poignées (*handle*) vers les éléments, aucun élément ne bougera vraiment, seuls les pointeurs bougent. Il faut faire un peu d'arithmétique de pointeurs pour décider si on doit aller vers l'avant ou vers l'arrière.

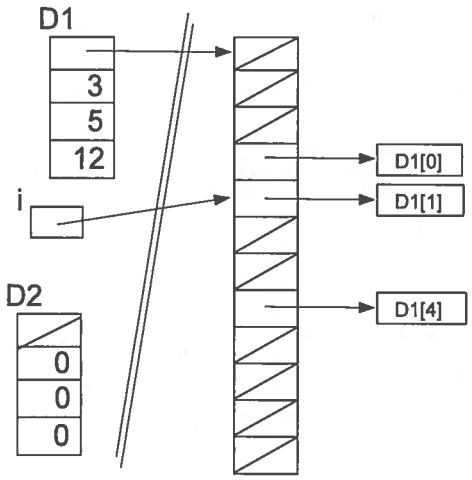
La description de la partie utile de la class `deque` vous est donnée ici, ainsi qu'une illustration d'un `deque` vide et d'un autre de dimension 5 (qui inclut une réserve de 3 éléments avant le début et de 4 éléments à la fin). On voit aussi la représentation d'un itérateur de `deque` "i" qui donne la position de l'élément numéro 1. Codez ces trois fonctions à la page suivante.

```
template <typename TYPE>
class deque{
private:
    TYPE** TAB;
    size_t AVANT,DIM,CAP;

public:
    class iterator;

    void pop_back();
    void pop_front();
    void erase(iterator);
    ...
};

template <typename TYPE>
class deque<TYPE>::iterator{
private:
    TYPE** REP;
public:
    ...
};
```



```
template <typename TYPE>
void deque<TYPE>::pop_back(){
```

```
template <typename TYPE>
void deque<TYPE>::pop_front(){
```

```
template <typename TYPE>
void deque<TYPE>::erase(typename deque<TYPE>::iterator i){
```

Question 3 – Utilisation des conteneurs de la SL (25 points)

Dans notre premier travail pratique (vous vous souvenez du Sudoku?), on a utilisé un `deque` pour représenter la grille et un `map` de `set` de la SL pour manipuler les contraintes :

```
map<int,set<int>> contraintes_globales;
```

Par exemple, les contraintes associées à la position 50 de la grille forment l'ensemble suivant :

`{5,14,23,30,31,32,39,40,41,45,46,47,48,49,51,52,53,59,68,77}`.

On veut maintenant optimiser un peu le code. On constate qu'il n'est pas obligatoire de traiter toutes les contraintes d'une position donnée, mais uniquement (1) celles qui se situent avant la position courante et (2) celles pour lesquelles les cases de la grille originale sont vides. Ainsi, dans les appels récursifs, quand on arrivera à la position 50 de la grille illustrée ici, l'ensemble à considérer sera `{14,23,30,32,40,41,48}`. En effet, la liste des possibilités pour une position donnée n'est pas toujours `{1,2,3,4,5,6,7,8,9}`, car on peut tenir compte des valeurs déjà connues de la grille. Par exemple, pour la position 50, on pourra toujours exclure 1, 2, 5, 6, 7 et 8. Quand viendra le temps de faire la liste des possibilités pour la case 50, on pourra partir de l'ensemble précalculé `{3,4,9}` et vérifier en plus les cases `{14,23,30,32,40,41,48}` pour lesquelles on a essayé des valeurs à date. Selon ces valeurs, qui ne sont plus à zéro, on pourra encore réduire le nombre de possibilités de la case 50.

Le conteneur `contraintes_globales` mentionné ci-dessus est déjà déterminé quand vous lisez un nouveau Sudoku. C'est à ce moment qu'il faut créer deux nouveaux conteneurs qui contiendront les contraintes réduites en fonction de la grille qui vient d'être lue, et les possibilités de base pour chacune des cases. Le plus simple est de le faire dans la fonction qui construit le Sudoku à partir d'une chaine de caractères. Voici la nouvelle déclaration de la classe Sudoku. On vous donne une partie de cette fonction, soit celle qui construit la grille : du code que vous aviez déjà pour le premier travail. Il vous reste à écrire le code qui construit le `vector` contenant les valeurs de base possibles pour chaque case de la grille (comme le `{3,4,9}` de la case 50 ci-dessus) et les contraintes pour chaque case (comme le `{14,23,30,32,40,41,48}` ci-dessus). Pour constituer ce nouveau conteneur, on n'a qu'à utiliser la grille et les contraintes globales. Pour les contraintes spécifiques à une grille, on a choisi d'utiliser des vecteurs plutôt que des `map` pour plus d'efficacité puisque l'accès à un `set` donné est O(1) dans un `vector<set<int>>`, mais O(log n) dans un `map<int,set<int>>`.

```
class sudoku{
private:
   static map<int,set<int>> contraintes globales;

   deque<int> grille;              //la grille de ce sudoku
   vector<set<int>> contraintes;   //contraintes spécifiques à cette grille
   vector<set<int>> possibles;     //les possibles précalculés pour cette grille
   ...
public:
   ...
   sudoku(const string&);
   ...
  };
```

				4	2			3
9			7					
								1
				2		5		
			1				8	
7	5	8		6			1	
			4			8		
	2		9		6	4		5
	4	3			7			

0	1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16	17
18	19	20	21	22	23	24	25	26
27	28	29	30	31	32	33	34	35
36	37	38	39	40	41	42	43	44
45	46	47	48	49	50	51	52	53
54	55	56	57	58	59	60	61	62
63	64	65	66	67	68	69	70	71
72	73	74	75	76	77	78	79	80

```
sudoku::sudoku(const string& S){
   init_contraintes();  //crée les contraintes globales au besoin
   //créer la grille à partir de la chaine de caractères S
   for(const auto& c : s)
      if(isdigit(c))grille.push_back(c-'0');
      else grille.push_back(0);
    grille.resize(81);

   //déterminer dans le conteneur "possibles" les valeurs possibles de base
   //pour chacune des 81 cases de la grille à partir des contraintes globales
   set<int> tous_les_possibles={1,2,3,4,5,6,7,8,9};
   possibles.resize(81);
   for(int i=0;i<81;++i){









   }

   //déterminer dans le conteneur "contraintes" les contraintes spécifiques
   //à cette grille sur la base des cases vides de la grille
   //et des contraintes globales
   contraintes.resize(81);
   for{int i=0;i<81;++i){









   }
}
```

Question 4 – Quelques éléments théoriques / techniques (25 points)

(1) **Complexité algorithmique** Donnez la complexité algorithmique des opérations suivantes, en expliquant sommairement l'algorithme utilisé pour réaliser la fonction qui justifie cette complexité.

a. localiser le ième élément d'un deque: O(_____)

b. localiser le ième élément d'un map: O(_____)

c. localiser l'élément X dans les champs `first` d'un map: O(_____)

d. localiser l'élément X dans les champs `second` d'un map: O(_____)

(2) **Effet de bord et retour de fonction** On sait qu'une fonction peut avoir un effet de bord et/ou un résultat en C++. Décrivez l'effet de bord et le résultat de chacune des fonctions données ci-dessous (ou indiquez-le s'il n'y en a pas)

	Effet de bord	Résultat
l'opérateur -- préfixé (comme `--i`):		
l'opérateur -= des int:		
l'opérateur >> des int: (comme i>>3)		
l'operateur << des flots (comme cout<<i;)		

(3) On désire itérer à l'envers sur l'ensemble des éléments d'un conteneur C de la SL. On peut le faire avec un iterator ou avec un reverse_iterator. Complétez le code des deux boucles suivantes qui font cette itération, l'une avec l'iterator `i` et l'autre avec le reverse_iterator `ri`.

```
for(                                                    ){


   cout<<*i<<endl;
   }

for(                                                    ){


   cout<<*ri<<endl;
   }
```

[FIN DE L'EXAMEN]