Département d'informatique
Examen intratrimestriel
Hiver 2016

IFT339

Structures de données

vendredi 26 février 2016
8 h 30 à 10 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. Vous n'avez droit qu'au **Résumé C++** jaune. Le surveillant vous fournira des feuilles blanches pour vos brouillons.

Identifiez-vous lisiblement ci-dessous :

Nom, prénom :	
Signature :	

1:	/8
2:	/8
3:	/8
4:	/8
	/32

Question 1 – Divers (8 points)

Votre matricule:

m7	m6	m5	m4	m3	m2	m1	m0

Arithmétique des types entiers (dans des int16_t). Indiquez s'il y a dépassement de capacité (oui ou non)

e				+			a		=				
	m2	m4	m7		m1	m6		m3					

dépassement?(o/n) _____

e				-			a		=				
	m2	m4	m7		m1	m6		m3					

dépassement?(o/n) _____

Comment pouvez-vous reconnaître qu'un algorithme est de complexité algorithmique...

a) O(n)?

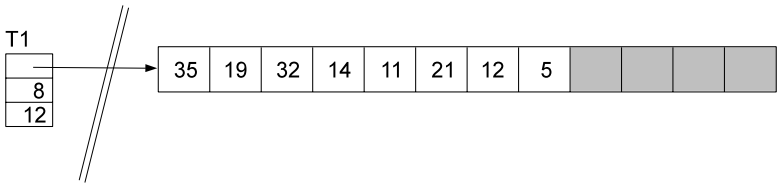
b) O(log n)?

c) O(1)?

Question 2 – Implantation d'un vector (8 points)

On explore ici une représentation différente du vector. On dispose des mêmes informations que ce qu'on a utilisé en laboratoire, mais codées de façon différente. Il y a trois membres pour la représentation. Un pointeur vers le début du tableau dynamique, un size_t qui donne la dimension logique (le size()) et un size_t qui donne la capacité totale, la dimension physique. On vous demande de reprendre la fonction reserve qui permet d'augmenter la capacité totale, ou de la réduire. Comme pour le laboratoire, si cette fonction est appelée avec une valeur inférieure à la capacité courante, elle ne doit pas la réduire. Par contre, par cohérence, elle doit tenir compte d'une demande de modifier la capacité qui la ramènerait sous la dimension. En effet, du point de vue de l'usager, la dimension ne peut pas être inférieure à la capacité.

```
template <typename TYPE>
class vector{
private:
   TYPE*  DEBUT;
   size_t DIM;
   size_t CAP;
public:
   ...
   void reserve(size_t);
   ...
};
```

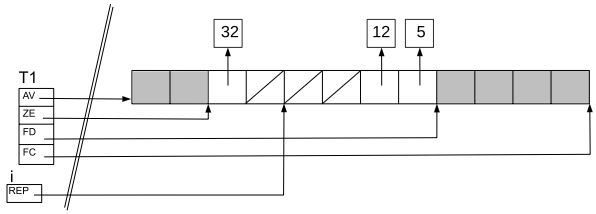



```
template <typename TYPE>
void vector<TYPE>::reserve(size_t nCAP){
```

Question 3 – Insérer dans un deque paresseux (8 points)

On a implanté dans le deuxième laboratoire un `deque` paresseux ("*lazy deque*") qui attend le plus tard possible avant d'instancier l'objet de type TYPE qu'une entrée doit contenir. On sait que l'insertion au début ou à la fin dans une telle structure prend un temps O(1) amorti, mais que l'insertion à un endroit arbitraire du deque, comme pour le vector, prend un temps O(n). Par contre, dans un deque, on peut améliorer un peu la situation par rapport au vector. Selon que l'insertion doit se faire plus près du début ou de la fin, on peut en profiter pour faire un push_front au lieu du push_back, pour ensuite décaler les éléments. Codez cette fonction insert, qui reçoit un itérateur en paramètre. Évidemment, on pourrait améliorer encore plus en vérifiant de quel côté il y a de l'espace de disponible, mais on ne fera pas ça ici. Une partie du code vous est donné, soit celle qui décide si on doit insérer à gauche ou à droite. Bien sûr, le résultat final du point de vue de l'usager sera le même. N'oubliez pas que cette fonction retourne la position de l'objet qui vient d'être inséré (le `return` est déjà codé pour vous!). Le reste est à faire.

```
template <typename TYPE>
class deque{
private:
    TYPE **AVANT,**ZERO,**FINDIM,**FINCAP;

public:
    class iterator;
    ...
    iterator insert(iterator,const TYPE&);
   ...
};
```

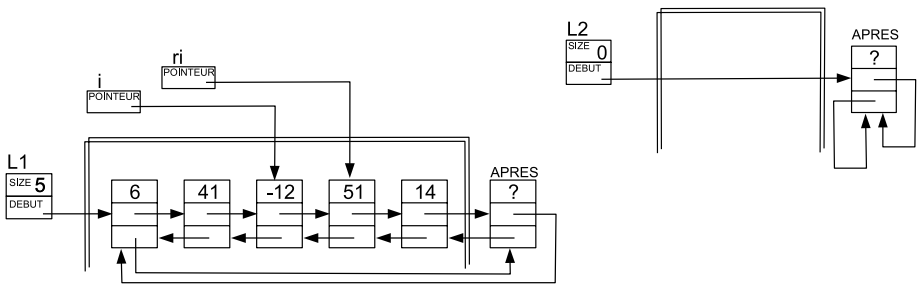



```
template <typename TYPE>
deque<TYPE>::iterator deque<TYPE>::insert(iterator i,const TYPE& VAL){
   size_t gauche=i.REP-ZERO;
   size_t droite=FINDIM-i.REP;
   if(gauche<droite){
      //inserer a gauche









      }
   else{
      //inserer a droite









      }
   return i;
   }
```

Question 4 – Un algorithme du conteneur *list* (8 points)

On a implanté une liste comme celle de la bibliothèque normalisée dans le troisième labo. On y a codé les algorithmes reverse, splice et sort. Il y a un quatrième algorithme spécialisé pour la liste, soit le resize. Celui-ci reçoit un size_t en paramètre et au retour la liste doit être devenue de cette dimension. Si on doit l'agrandir, on peut demander d'insérer à la fin dans les nouvelles positions une valeur spécifique. Sinon, la valeur par défaut est celle déterminée par le type de l'objet. On utilise la même représentation que dans le laboratoire, avec une cellule de queue qui permet d'unifier la position de la fin avec les autres positions. On en voit des illustrations ci-contre.

```
template <typename TYPE>
class list{
private:
    struct cellule{
        TYPE CONTENU;
        cellule *SUIV,*PREC;
        ...
        };
    cellule *DEBUT;
    cellule APRES;
    size_t SIZE;
public:
   ...
   void resize(size_t,const TYPE& = TYPE());
   ...
};
```




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




}
```