Epreuve écrite avec convocation :
Algorithmique et Structures de données

Semestre S5 d'automne 2013/2014 - durée : 1h30

L3 Mathématiques - DUAS - Magistère (Université de Strasbourg)

Ce sujet comporte 4 pages et 2 parties indépendantes. On s'attachera à soigner la présentation du code afin qu'il soit le plus lisible possible. Il ne suffit en aucun cas d'écrire le code d'une fonction, il faut expliquer (i.e. commenter) les choix faits pour l'implantation. Il est de plus indispensable de respecter les notations données dans l'énoncé.

1 Echauffement

On rappelle une définition (et une implantation possible) des listes non vides sous forme d'une classe. Afin de pouvoir tester notre implantation, on programme également une fonction d'affichage (par surcharge de l'opérateur $<<$).

```
#include<iostream>
using namespace std;
#define N 10

class listeNV // listes non vides
{
  int v;
  listeNV *suiv;
public:
listeNV(int v, listeNV *s);

  void plus1();
  int somme();
  friend ostream & operator<<(ostream &o, listeNV l);
} ;

listeNV::listeNV(int i, listeNV *l)
{
  v=i;
  suiv=l;
}

int main()
{
  listeNV *l = new listeNV(0,NULL);
  cout<<*l<<endl;
  for (int i=1; i<=5; i++) l=new listeNV(i,l);
```

```
  cout<<*l<<endl;
  l->plus1();
  cout<<*l<<endl;
  cout<<"somme = "<<l->somme()<<endl;
  return 0;
}
```

Question 1 Programmer la méthode `plus1` (qui ajoute 1 à chaque élement de la liste). On procèdera de manière récursive avec des effets de bord (le type de retour de la méthode est `void`).

```
void listeNV::plus1()
```

Question 2 Programmer de manière récursive une méthode `somme` qui fait la somme de tous les éléments d'une liste.

```
int listeNV::somme()
```

Question 3 Programmer de manière itérative (avec une boucle `while`) l'affichage d'une liste à l'aide de la surcharge de l'opérateur <<. En déduire ce que la fonction `main` affiche à l'écran.

```
friend ostream & operator<<(ostream &o, listeNV l);
```

2 Problème : arbres binaires parfaits partiellement ordonnés

Un arbre binaire est dit *parfait* lorsque tous ses niveaux sont complètement remplis, sauf éventuellement le dernier niveau et dans ce cas les noeuds (qui sont des feuilles) du dernier niveau sont groupés le plus à gauche possible. L'arbre ci-dessous est un arbre parfait. Un arbre

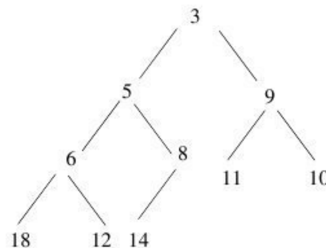

Figure 1 – Exemple de tas

binaire parfait de taille n peut se représenter de façon très compacte dans un tableau de taille au moins égale à $n + 1$ (la première case n'est pas utilisée). La représentation est la suivante :

La racine de l'arbre est en $T[1]$ et si un noeud se trouve en $T[i]$, son fils gauche est en T[2*i] et son fils droit en T[2*i+1].

Question 4 Dessiner un tableau de taille 10 permettant de représenter l'arbre binaire de la figure 1.

Question 5 Dessiner l'arbre binaire parfait qui correspond au tableau suivant (la première case - d'indice 0 - est vide) :

	3	12	7	18	16	22	13

On considère le canevas de programme de la figure 2. Vous répondrez aux questions suivantes en utilisant la classe (ses structures de données et ses méthodes) présentée dans ce morceau de programme.

Question 6 Expliquer les structures de données et les différentes méthodes définies dans la figure 2 et leurs rôles. On précisera notamment le sens des arguments ainsi que les préconditions utiles qu'il pourrait être nécessaire de rajouter dans le code.

Un arbre binaire parfait est dit *partiellement ordonné* si l'étiquette de tout noeud est inférieure à celle de ses fils. Le minimum de l'ensemble représenté par un tel arbre est donc situé la racine. Un tel est appelé *un tas*.

Question 7 L'arbre dessiné à la question **5**, qui est parfait, vérifie-t-il la propriété d'être partiellement ordonné ?

Dans la suite, on travaille avec des arbres binaires parfaits partiellement ordonnés (aussi appelés "tas").

2.1 Ajouter un élement dans le tas

On commence par rajouter l'élément x à insérer comme feuille au dernier niveau de l'arbre puis on échange l'étiquette du noeud portant x avec celle de son père jusqu'à ce que, pour respecter la structure d'arbre binaire parfait partiellement ordonné, x ne soit pas inférieur à l'étiquette du père du noeud qui le porte.

Question 8 Faire un dessin des étapes successives de l'insertion du nombre 4 dans l'arbre de la figure 1.

Question 9 Programmer une fonction **insertion** qui réalise cette opération

```
void tas::insertion(int i)
{ ... }
```

2.2 Supprimer l'élement minimum du tas

L'idée est de supprimer la dernière feuille du dernier niveau après avoir recopié son étiquette à la racine, puis, toujours pour respecter la structure d'arbre binaire parfait partiellement ordonné, de faire redescendre cette valeur en la comparant au contenu de ses fils, par un processus inverse de celui de l'insertion.

Question 10 Faire un dessin des étapes sucesssives à réaliser lors de la suppression du minimum dans l'arbre donné en exemple dans la figure 1. Lors de la descente d'une valeur dans l'arbre, on privilégiera la branche avec l'étiquette minimale lorsqu'on a le choix du côté de l'arbre où faire descendre l'élément courant.

Question 11 Programmer une fonction **suppression_min** qui supprime le minimum (i.e. la racine) dans un arbre binaire parfait partiellement ordonné.

```
void tas::suppression_min()
```

2.3 Un programme de tri

Question 12 Expliquer comment les opérations programmées précédemment pourraient être utilisées pour trier n nombres par ordre croissant. On supposera que les données fournies en entrée le sont sous la forme d'un tableau de taille n.

Question 13 Programmer un tel algorithme avec les fonctions construites jusqu'ici.

```
#include<iostream>
using namespace std;
#define N 10

class tas
{ int *t; // tableau qui sera alloué dynamiquement
  int n; // taille courante du tas
  int max; // taille max. possible du tas
public:
  tas(int taille);
  int etiquette(int i);
  int racine();
  int is_vide(int p);
  int fg(int i);
  int fd(int i);
  int pere(int i);

  void insertion(int v);
  void suppression_min();
  friend ostream &operator<<(ostream &o, tas x);
} ;

tas::tas(int taille);
// initialise les variables de la classe !!!

int tas::etiquette(int i);
// renvoie l'élément de l'arbre représenté dans la case i du tableau

int tas::racine();
// retourne la racine d'arbre représenté par le tableau

int tas::is_vide(int p);
// le sous-arbre a partir de p est-il vide ?

int tas::fg(int i);
// calcule la position du fils gauche dans le tableau
// (renvoie 0 s'il n'existe pas)

int tas::fd(int i);
// calcule la position du fils droit dans le tableau
// (renvoie 0 s'il n'existe pas)

int tas::pere(int i);
// calcule la position du père de i dans le tableau

ostream &operator<<(ostream &o, tas x);
// affiche l'arbre tel qu'il est représenté dans la mémoire
// sous forme de tableau
```

FIGURE 2 – Déclaration des opérations de base sur les tas - Implantation à complèter