Algorithmique et programmation
Piles - files – Solutions

```
//listeInt.h
#include <cstddef>
using namespace std;

class NoeudInt
{
  public:
    int donnee;
    NoeudInt *svt;
    NoeudInt(int d = 0, NoeudInt *s = NULL)
      :donnee(d), svt(s) {};
};
```

STACK

```
// stack.h
#include "listeInt.h"

class Stack
{
  public:
    Stack();
    Stack(const Stack &);
    ~Stack();
    bool empty() const;
    bool push(int);
    bool pop();
    bool pop(int &);
    bool top(int &) const;
  private:
    Node *stackTop;
};
```

```
//stack.cpp
#include "stack.h"
#include <cstdlib>
using namespace std;

Stack::Stack(): stackTop(NULL) {};

Stack::Stack(const Stack &s)
{
  if (!s.stackTop)
    stackTop = NULL;
  else
  {
    Node *cur;
    stackTop = new Node(s.stackTop->data);
    cur = stackTop;
    for(Node *ptr = s.stackTop->next; ptr; ptr = ptr->next)
      cur = cur->next = new Node(ptr->data);
```

```
  }
}

Stack::~Stack()
{
  while (pop());
}

bool Stack::empty() const
{
  return !stackTop;
}

bool Stack::push(int d)
{
  Node *tmp = new Node(d, stackTop);

  if (!tmp)
    return false;
  stackTop = tmp;
  return true;
}

bool Stack::pop()
{
  Node *tmp;

  if (empty())
    return false;
  tmp = stackTop;
  stackTop = stackTop->next;
  delete tmp;
  return true;
}

bool Stack::pop(int &d)
{
  Node *tmp;

  if (!top(d))
    return false;
  pop();
  return true;
}

bool Stack::top(int &d) const
{
  if (empty())
    return false;
  d = stackTop->data;
  return true;
}
```

```
//queue.h
#include "listeInt.h"

class Queue
{
  public:
    Queue();
    Queue(const Queue& q);
    ~Queue();
    bool empty() const;
    bool insert(int);
    bool get();
    bool get(int &);
    bool front(int &) const;
  private:
    Node *backPtr; // backPtr pointe vers le dernier element
                   // la queue est une liste cyclique, le next de
                   backPtr   // pointe vers le premier element de la liste
};
```

```
//queue.cpp
#include "queue.h"
#include <cstdlib>
using namespace std;

Queue::Queue()
{
  backPtr = NULL;
}

Queue::Queue(const Queue& q)
{
  if (q.empty())
    backPtr = NULL;
  else
  {
    Node *curQ, *cur;

    curQ = q.backPtr;
    backPtr = new Node(curQ->data);
    cur = backPtr;
    curQ = curQ->next;
    while (curQ != q.backPtr)
    {
      cur = cur->next = new Node(curQ->data);
      curQ = curQ->next;
    }
    cur->next = backPtr;
  }
}

Queue::~Queue()
{
  while (!empty())
    get();
}

bool Queue::empty() const
```

```
{
  return !backPtr;
}

bool Queue::insert(int n)
{
  Node *ptr = new Node(n); // creation du noeud

  if (ptr == NULL)
    return false;
  if (empty())
    ptr->next = ptr;
  else
  {
    ptr->next = backPtr->next; // insertion en tete
    backPtr->next = ptr; // et on deplace le cycle de un element
  }
  backPtr = ptr;
  return true;
}

bool Queue::get()
{
  if (empty())
    return false;
  Node *ptr = backPtr->next; // se place sur le suppose premier element
  if (backPtr == ptr) // si la queue ne contenait qu'un element
    backPtr = NULL; // maintenant y en a plus
  else
    backPtr->next = ptr->next; // sinon on saute le premier element
  delete ptr; // et on le supprime
  return true;
}

bool Queue::get(int &n)
{
  if (!front(n))
    return false;
  get();
  return true;
}

bool Queue::front(int &n) const
{
  if (empty())
    return false;
  n = backPtr->next->data;
}
```

Piles

Exercice 1.

Ecrire une fonction non récursive qui affiche les éléments d'une liste d'entiers dans l'ordre inverse de leur apparition dans la liste sans modifier la liste initiale mais en utilisant une pile.

```
#include <iostream>
#include "stack.h"
using namespace std;

void afficherEnvers(NoeudInt *l)
{
  Stack s;
  int d;

  while(l != NULL)
  {
    s.push(l->donnee);
    l = l->svt;
  }
  while(s.pop(d))
    cout << d << " ";
  cout << endl;
}

void afficherListe(NoeudInt *l)
{
  while(l != NULL)
  {
    cout << l->donnee << " ";
    l = l->svt;
  }
  cout << endl;
}

main()
{
  int tab[] = {5, 4, 3, 2, 1};
  NoeudInt *l = NULL;

  for(int i = 0; i < 5; i++)
    l = new NoeudInt(tab[i], l);

  afficherListe(l);

  afficherEnvers(l);
}
```

Exercice 4.

Dans une expression en notation postfixe (ou notation polonaise inverse NPI), les opérateurs sont placés après les opérandes rendant les parenthèses superflues. On demande d'écrire une fonction qui retourne la valeur d'une expression en NPI passée en paramètre dans une chaîne de caractères. Les opérandes de l'expression seront des variables de type int représentées par une lettre minuscule, la fonction recevra donc également en paramètre un tableau contenant les valeurs des variables de l'expression: l'élément 0 contenant la valeur de a, l'élément 0 celle de b, …
On considère que l'expression ne fait intervenir que les opérateurs +, -, * et / (division entière) et qu'elle est syntaxiquement correcte.

Exemple.

La valeur de l'expression NPI ab+ca*- avec a=2, b=7 et c=3 vaut 3.

Pour évaluer une expression en NPI, il faut lire l'expression de gauche à droite. Quand on rencontre une variable, on empile sa valeur sur une pile initialement vide. Quand on rencontre un opérateur, on enlève les deux valeurs au sommet de la pile, on leur applique l'opérateur et on empile le résultat. A la fin du parcours de l'expression, il ne reste qu'une valeur dans la pile qui est la valeur de l'expression.

```
int evalNPI(char *expr, int val[])
{
  Stack s;
  int oper1, oper2, res;

  while (*expr)
  {
    if (*expr >= 'a' && *expr <= 'z')
      s.push(val[*expr - 'a']);
    else
    {
      s.pop(oper1);
      s.pop(oper2);
      switch (*expr)
      {
        case '+': s.push(oper2 + oper1);
                  break;
        case '-': s.push(oper2 - oper1);
                  break;
        case '*': s.push(oper2 * oper1);
                  break;
        case '/': s.push(oper2 / oper1);
                  break;
      }
    }
    expr++;
  }
  s.pop(res);
  return res;
}
```

Exercice 5.

Ecrire une fonction qui évalue une expression en notation infixe passée en paramètre dans une chaîne de caractères et ne faisant intervenir que les opérateurs +, -, * et / et des variables entières représentées par des lettres minuscules. On supposera que toutes les sous-expressions de l'expression sont toujours placées entre parenthèses. La fonction recevra comme paramètre supplémentaire un tableau contenant les valeurs des variables de l'expression: l'élément 0 contenant la valeur de a, l'élément 0 celle de b, …

Pour évaluer une telle expression, il faut utiliser deux piles: une pile de valeurs et une pile d'opérateurs (on peut coder les opérateurs +=1, -=2, *=3, /=4 pour pouvoir utiliser une pile d'entiers). On parcourt l'expression de gauche à droite. Quand on rencontre une variable on empile sa valeur sur la pile des valeurs. Quand on rencontre un opérateur, on l'empile sur la pile des opérateurs. Quand on recontre une parenthèse fermante, on enlève les deux valeurs au sommet de la pile des valeurs et l'opérateur au sommet de la pile des opérateurs. On applique l'opérateur aux deux valeurs et on empile le résultat sur la pile des valeurs. Une fois l'expression parcourue, sa valeur est l'unique élément de la pile.

```
int evalInfixe(char *expr, int val[])
{
  Stack sVal;
  Stack sOp;
```

```
  int op, oper1, oper2, res;

  while (*expr)
  {
    if (*expr >= 'a' && *expr <= 'z')
      sVal.push(val[*expr - 'a']);
    else
      switch (*expr)
      {
        case '+': sOp.push(1);
                  break;
        case '-': sOp.push(2);
                  break;
        case '*': sOp.push(3);
                  break;
        case '/': sOp.push(4);
                  break;
        case ')':
          sVal.pop(oper1);
          sVal.pop(oper2);
          sOp.pop(op);
          switch (op)
          {
            case 1: sVal.push(oper2 + oper1);
                    break;
            case 2: sVal.push(oper2 - oper1);
                    break;
            case 3: sVal.push(oper2 * oper1);
                      break;
            case 4: sVal.push(oper2 / oper1);
                      break;
          }

      }
    expr++;
  }
  sVal.pop(res);
  return res;
}
```

Files

Exercice 3.

Ecrire une fonction qui inverse une file d'entiers qui lui est passée comme paramètre.

On demande de ne pas utiliser de tableau ou de liste de travail pour effectuer l'inversion mais d'utiliser plutôt une pile. Il existe en effet une méthode très simple pour inverser une file en utilisant une pile.

On suppose disponibles les classes Queue et Stack dont les parties publiques sont données ci-dessous et pour lesquelles vous ne devez donner ni la partie privée ni le code des méthodes.

```
class Queue
{
  public:
    Queue();
    Queue(const Queue &q);
    ~Queue();
    bool empty();
```

```
    void insert(int n);
    int get();
};

class Stack
{
  public:
    Stack();
    Stack(const Stack &s);
    ~Stack();
    bool empty();
    void push(int n);
    int pop();
};


#include <iostream.h>
#include "stack.h"
#include "queue.h"

void inverser(Queue &q)
{
  Stack s;
  while ( !q.empty() )
    s.push(q.get());
  while ( !s.empty() )
    q.insert(s.pop());
}

main()
{
  Queue q;
  q.insert(1);
  q.insert(2);
  q.insert(3);
  q.insert(4);
  inverser(q);
  while ( !q.empty() )
    cout << q.get() << endl;
}
```