3.3. Pilas

Una Pila es una estructura de datos lineal que sólo permite insertar, eliminar y consultar elementos en uno de sus extremos, que denominamos *cima*. Es, por tanto, una estructura de tipo **LIFO** (Last In, First Out), en la cual el último elemento que entra es el primer elemento que sale.

La siguiente especificación de la clase `stack` resume las principales operaciones básicas del tipo de datos `stack` (pila) implementada en la STL (Standard Template Library) de C++.

```
template <class T> class stack {
// Tipo de módulo: datos
// Descripción del tipo: Estructura lineal que contiene elementos
// de tipo T, que permite añadir, consultar y eliminar elementos
// sólo en uno de sus extremos.

public:

// Constructores

stack();
/* Pre: cierto */
/* Post: Construye una pila vacía. */

stack(const queue &original);
/* Pre: cierto */
/* Post: Construye una pila que es una copia de "original". */


// Modificadores

void push(const T& x);
/* Pre: El parámetro implícito es P. */
/* Post: El parámetro implícito es p, donde p es igual a P
con x añadido como último elemento (en la cima). */

void pop();
/* Pre: El parámetro implícito es P. P no es una pila vacía. */
/* Post: El parámetro implícito es p, donde p es igual a P sin su
 último elemento (el que estaba en la cima). */

T& top();
/* Pre: El parámetro implícito no es una pila vacía. */
/* Post: El resultado es el último valor del parámetro implícito,
i.e. el elemento situado en la cima de la pila. */
```

```
// Consultores

const T& top() const;
/* Pre: El parámetro implícito no es una pila vacía. */
/* Post: El resultado es el último valor del parámetro implícito,
i.e. el elemento situado en la cima de la pila. */

bool empty() const;
/* Pre: cierto */
/* Post: El resultado indica si el parámetro implícito es una
pila vacía. */

int size() const;
/* Pre: cierto */
/* Post: El resultado es el número de elementos del parámetro implícito. */

};
```

3.3.1. Ejemplos de Uso

Los compiladores comprueban si los programas contienen errores sintácticos. Normalmente, la omisión de un símbolo, por ejemplo un paréntesis o el marcador del inicio de un comentario, puede generar muchos mensajes de diagnóstico del compilador que no sirven para identificar el verdadero error que contiene el programa. Una herramienta útil en esta situación es un programa que compruebe que todo paréntesis, llave, corchete o marcador de inicio de un bloque abierto se cierra adecuadamente. Por ejemplo, la secuencia `[()]` es correcta, pero la secuencia `[(])` no es sintácticamente correcta. En la lista de ejercicios de la sesión de laboratorio correspondiente a este tema encontraréis un problema que requiere construir un programa sencillo que comprueba si una secuencia de paréntesis y corchetes es sintácticamente correcta utilizando una pila de caracteres. El programa debe ser lineal, realizar como máximo un solo recorrido de la secuencia de caracteres, e informar de la posición donde se ha encontrado el primer error si la secuencia no es sintácticamente correcta.

Evaluación expresión aritmética en notación postfija

La evaluación de expresiones aritméticas en notación infija requiere tener en cuenta las distintas prioridades de los operadores multiplicativos y aditivos, las reglas de asociatividad de estos operadores, y el orden de evaluación de argumentos implicado por el uso de paréntesis. Evaluar una expresión aritmética en notación postfija (también conocida como notación Polaca inversa) es sin embargo más sencillo, porque no es necesario conocer las reglas de precedencia entre los distintos tipos de operadores aritméticos. El tiempo necesario para evaluar una expresión aritmé-

tica en notación postfija es $O(n)$, donde n es el número de símbolos (i.e. números y operadores) que contiene la expresión aritmética.

El siguiente programa utiliza un pila de enteros para evaluar una expresión aritmética representada en notación postfija. Como en el ejemplo anterior, sólo es preciso recorrer una vez la secuencia de símbolos que representa la expresión aritmética. A medida que se van procesando estos símbolos, se utiliza la pila de enteros para almacenar resultados intermedios y para recuperarlos cuando deban ser usados como argumentos de operadores que se encuentra en posiciones posteriores de la expresión aritmética. Debido a que las operaciones de una pila de enteros tienen coste constante, el coste total del algoritmo es lineal, i.e. $O(n)$.

```
#include <iostream>
#include <string>
#include <stack>
using namespace std;

int eval(const string& expression) {
  // Pre: expression es una expresión aritmética formada por naturales
  // entre 0 y 9, + y *.
  stack<int> arg;
  for (int i = 0; i < expression.size(); ++i) {
    if ('0' <= expression[i] and expression[i] <= '9') {
       arg.push(expression[i] - '0');
    }
    else {
      int arg1 = arg.top();
      arg.pop();
      int arg2 = arg.top();
      arg.pop();
      int result;
      if (expression[i] == '*') result = arg1*arg2;
      else result = arg1+arg2;
      arg.push(result);
    }
  }
  return arg.top();
}

int main() {
  string exp;
  cin >> exp;
  cout << endl << eval(exp) << endl;
}
```

Notación infija a postfija

También se puede utilizar una pila para convertir una expresión aritmética en notación infija en una expresión aritmética equivalente en notación postfija. A continuación presentamos un programa que utiliza una pila de caracteres para realizar esta conversión cuando las expresiones sólo contienen números naturales, paréntesis, los operadores aritméticos $+$ y $*$, y cada subexpresión de la forma `A operador B` está rodeada por un par de paréntesis, es decir, se representa como `(A operador B)`.

```
#include <iostream>
#include <string>
#include <stack>
using namespace std;

string to_postfix(const string& infix) {
  string postfix;
  stack<char> oper;
  for (int i = 0; i < infix.size(); ++i) {
    if (infix[i] == ')') {
      postfix += oper.top(); // concatenación
      oper.pop();
    }
    else if (infix[i] == '*' or infix[i] == '+') {
      oper.push(infix[i]);
    }
    else if ('0' <= infix[i] and infix[i] <= '9') {
      postfix += infix[i];
    }
  }
  return postfix;
}

int main() {
  string inp;
  cin >> inp;
  cout << to_postfix(inp) << endl;
}
```

Para transformar $(A+B)$ en $AB+$, ignoramos los caracteres correspondientes a los paréntesis abiertos, escribimos A en forma postfija, almacenamos el operador $+$ en la pila, escribimos B en notación postfija, y cuando encontramos el carácter correspondiente al paréntesis cerrado, sacamos el operador $+$ de la pila y lo escribimos.

El operador += asigna a la cadena de caracteres colocada a su izquierda el resultado de concatenar la cadena de caracteres colocada a su izquierda con el carácter o cadena de caracteres

colocada a su derecha.

Recorrido en profundidad de un grafo

Un *grafo dirigido* G es un par (V,E), donde V es un conjunto finito que representa los vértices de G, y E es una relación binaria definida sobre V que representa los ejes de G.

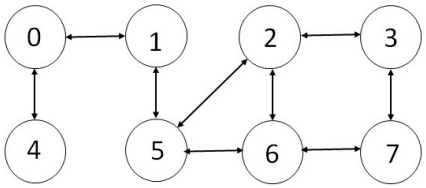

Figura 3.1: Grafo dirigido con $V = \{0,1,2,3,4,5,6,7\}$ y ejes representados en la figura. El resultado del algoritmo de recorrido en profundidad `dft` a partir del vértice 1 es 1, 5, 6, 7, 3, 2, 0, 4, suponiendo que la lista de adyacencia es la que se muestra en el cuadro 3.1.

Lista de Adyacencia	
Vértice	Vértices adyacentes
0	1, 4
1	0, 5
2	3, 5, 6
3	2, 7
4	0
5	1, 2, 6
6	2, 5, 7
7	3, 6

Cuadro 3.1: Vértices del grafo de la figura 3.1 seguidos por su lista de adyacencia.

La representación de un grafo $G = (V,E)$ mediante una *lista de adyacencia* consiste en un vector Ad de $|V|$ vectores[1], uno por cada vértice de G. Dado un vértice $u \in V$, la lista de adyacencia $Ad[u]$ contiene los vértices x tales que existe un eje de la forma (u,x) en E. Es decir, la lista de adyacencia de un vértice u contiene todos los vértices con los que u está conectado en G. La figura 3.1 muestra un ejemplo de grafo dirigido, que podría representarse mediante una lista de adyacencia como la que se muestra en el cuadro 3.1.

Dado un grafo $G = (V,E)$ y un vértice origen s de V, presentamos un algoritmo de *recorrido en profundidad* (`dft`) que explora sistemáticamente los ejes de G, con el objetivo de examinar todos los vértices de G que se pueden alcanzar a partir de s.

[1]Una lista de adyacencia se suele representar con un vector de listas, en lugar de un vector de vectores. Cuando veamos la estructura de datos `lista` en este curso, dentro de unas semanas, podréis adaptar esta definición de manera que cada $Ad[u]$ sea una lista de vértices en lugar de un vector de vértices.

Un algoritmo de recorrido en profundidad examina los vértices más profundos de un grafo en primer lugar, siempre que esto sea posible. Es decir, explora los ejes que salen del vértice *v* descubierto más recientemente. Cuando se han explorado todos los ejes que salen de *v*, el algoritmo retrocede (en inglés `backtracks`) e intenta explorar los ejes aún no explorados que salen del vértice a partir del cual se descubrió *v*. Este proceso continua hasta que se han examinado todos los vértices alcanzables a partir del vértice origen del recorrido.

La acción `dft` asume que el grafo está representado mediante una lista de adyacencia `t`. Utiliza un vector de booleanos `examined` para anotar los vértices que ya han sido examinados, y una pila `not_exp` para almacenar los vértices examinados cuyos ejes no han sido explorados. Inicialmente, todos los vértices se marcan como no examinados, y se inserta el nodo origen `s` en la pila. A medida que el algoritmo va descubriendo nuevos vértices alcanzables desde el vértice `s` los marca como examinados.

El bucle principal itera sobre los vértices que contiene la pila. Explora cada uno de sus ejes, comprueba si sus vértices adyacentes ya han sido examinados, y si no lo han sido, los marca como examinados y los añade a la pila. El algoritmo termina cuando no queda ningún vértice en la pila, es decir, ningún vértice cuyos ejes no hayan sido explorados.

```
#include <iostream>
#include <stack>
#include <vector>
using namespace std;

typedef vector<vector<int> > List_Ad;

void dft(const List_Ad& t, int s) {
  int n = t.size();
  vector<bool> examined(n,false);
  examined[s] = true;
  stack<int> not_exp;
  not_exp.push(s);
  while (not not_exp.empty()) {
    int act = not_exp.top();
    not_exp.pop();
    for (int i = 0; i < t[act].size(); ++i) {
      if (not examined[t[act][i]]) {
         examined[t[act][i]] = true;
         not_exp.push(t[act][i]);
      }
    }
    cout << act << endl;
  }
}
```

```
void read(List_Ad& t) {
  int n;
  cin >> n;
  t = List_Ad(n);
  for (int i = 0; i < n; ++i) {
    int m;
    cin >> m;
    t[i] = vector<int>(m);
    for (int j = 0; j < m; ++j) cin >> t[i][j];
  }
}

int main() {
  List_Ad a;
  read(a);
  int source;
  cin >> source;
  vector<int> d;
  dft(a,source);
}
```

3.4. Colas

Una cola es una estructura de datos lineal que permite: (1) insertar elementos sólo en uno de sus extremos, denominado *final*; (2) eliminar elementos sólo en el otro extremo, denominado *cabeza*; y (3) consultar únicamente su primer elemento, situado a la cabeza de la cola.

Una cola es, por tanto, una estructura de tipo FIFO (First In, First Out), en la cual el primer elemento que entra es el primer elemento que sale.

La siguiente especificación de la clase `queue` resume las principales operaciones básicas del tipo de datos `queue` (cola) implementada en la STL de C++.

```
template <class T> class queue {

// Tipo de módulo: datos
// Descripción del tipo: Estructura lineal que contiene elementos
// de tipo T, que permite añadir elementos sólo al final, y
// consultar y eliminar únicamente su primer elemento.

public:

// Constructores
```

```
queue();
/* Pre: cierto */
/* Post: Construye una cola vacía. */

queue(const queue &original);
/* Pre: cierto */
/* Post: Construye una cola que es una copia de "original". */


// Modificadores

void push(const T& x);
/* Pre: El parámetro implícito es P. */
/* Post: El parámetro implícito es p, donde p es igual a P
con x añadido como último elemento. */

void pop();
/* Pre: El parámetro implícito es P. P no es una cola vacía. */
/* Post: El parámetro implícito es p, donde P es igual a P sin
su primer elemento. */

T& front();
/* Pre: El parámetro implícito no es una cola vacía. */
/* Post: El resultado es el primer elemento del parámetro implícito. */


// Consultores

const T& front() const;
/* Pre: El parámetro implícito no es una cola vacía. */
/* Post: El resultado es el primer elemento del parámetro implícito. */

bool empty() const;
/* Pre: cierto */
/* Post: El resultado indica si el parámetro implícito es una
cola vacía. */

int size() const;
/* Pre: cierto */
/* Post: El resultado es el número de elementos del parámetro implícito. */

};
```

3.4.1. Ejemplos de Uso

Hay muchos algoritmos que utilizan colas para obtener soluciones de forma más eficiente, por ejemplo en teoría de grafos.

Los archivos que se envían para imprimir en una impresora se procesan en orden de llegada, es decir, se almacenan esencialmente en una cola.

Prácticamente cualquier cola en la vida real se comporta como la estructura de datos Cola, ya que se atiende en primer lugar al primero que llega.

Las llamadas telefónicas en grandes empresas se colocan generalmente en una cola cuando todos los operadores están ocupados.

En matemáticas existe un área de investigación denominada *teoría de colas* cuyo objetivo es calcular de manera probabilística cuánto deben esperar los usuarios de una cola, la longitud que puede llegar a tener una cola, y otras cuestiones relacionadas. La respuesta depende de la frecuencia con la que llegan los usuarios a la cola, y del tiempo que cuesta atender a cada usuario. Ambos parámetros suelen darse mediante funciones que representan distribuciones de probabilidad. Para problemas sencillos la solución puede calcularse de forma analítica, pero en casos complejos de resolver de forma analítica suelen utilizarse técnicas de simulación computacional.

Simulación de un sistema de colas

Presentamos un programa que simula una oficina de atención al público. En un momento dado existen n clientes a la espera, que identificamos por su orden de llegada (i.e. números del 1 a n). La oficina decide abrir m ventanillas para atender a estos clientes, teniendo en cuenta su orden de llegada. En primer lugar distribuye los clientes de forma equitativa entre las m colas, una por ventanilla. Y, una vez que los n clientes están distribuidos en las m colas, escribe el orden en el que son atendidos hasta que se vacían las colas de todas las ventanillas.

El programa utiliza un vector de m colas. En primer lugar, asigna los n clientes a las distintas colas. Este proceso se realiza asignando cada cliente i a una de las colas más cortas que existen en el momento de la asignación de i a una de la m colas. Si existen varias colas de longitud mínima, elige una de las colas de longitud mínima de forma aleatoria. La variable *min* representa la longitud de la cola más corta en cada momento de la asignación.

En segundo lugar, se simula el proceso de atención de los clientes que ya están distribuidos en las m colas correspondientes a cada ventanilla, suponiendo que el tiempo necesario para atender a cada cliente es el mismo.

```
#include <iostream>
#include <queue>
#include <vector>
using namespace std;

 void actualiza(int i, int& min, vector<queue<int> >& v) {
  vector<int> mas_cortas;
  for (int i = 0; i < v.size(); ++i) {
    if (v[i].size() == min) mas_cortas.push_back(i);
```

```
  }
  int n = mas_cortas.size();
  if (n == 1) ++min;
  if (n > 1) {
    int j = random() % n;
    v[mas_cortas[j]].push(i);
  }
  else v[mas_cortas[0]].push(i);
}



void atiende(vector<queue<int> >& v, bool& alguno) {
  alguno = false;
  for (int i = 0; i < v.size(); ++i) {
    if (not v[i].empty()) {
      cout << v[i].front() << " -> ventanilla " << i+1 << endl;
      v[i].pop();
      if (not v[i].empty()) alguno = true;
    }
  }
  cout << endl;
  if (alguno)   cout << endl << "Nuevo turno" << endl;
}

int main() {
  int n;
  cin >> n; // número de clientes
  int m;
  cin >> m; // número de colas
  vector<queue<int> > v(m);
  int min = 0; // longitud cola más corta
  for (int i = 0; i < n; ++i) actualiza(i+1,min,v);
  bool alguno = n > 0;
  while (alguno) atiende(v,alguno);
}
```

Recorrido en anchura de un grafo

Dado un grafo $G = (V,E)$ y un vértice origen s de V, el algoritmo de *recorrido en anchura* (`bft`) que presentamos explora sistemáticamente los ejes de G con el objetivo de descubrir todos los vértices que se pueden alcanzar a partir de s. Además, calcula la distancia a s de cada vértice alcanzable (i.e. el menor número de ejes que es preciso recorrer para alcanzar dicho vértice

partiendo de s).

El algoritmo de recorrido en anchura también se denomina recorrido por niveles, porque examina todos los vértices que se encuentran a distancia k de s antes de examinar todos los vértices que se encuentran a distancia $k+1$.

La acción `bft`, presentada a continuación, asume que el grafo dirigido que examina el algoritmo está representado mediante una lista de adyacencia t, utiliza un vector de booleanos `examined` para anotar los vértices que ya han sido examinados, un vector de enteros `distance` para almacenar la distancia de s a cada vértice, y una cola para representar los vértices examinados cuyos ejes no han sido explorados. Inicialmente, todos los vértices se marcan como no examinados y con distancia -1. A medida que el algoritmo va descubriendo nuevos vértices alcanzables desde el vértice `s` los marca como examinados, anota su distancia a `s`, y los guarda en la cola.

El bucle principal itera sobre los vértices que contiene la cola. Explora cada uno de sus ejes, comprueba si sus vértices adyacentes ya han sido examinados, y si no lo han sido, los marca como examinados y los añade a la cola. El algoritmo termina cuando no queda ningún vértice en la cola, es decir, ningún vértice cuyos ejes no hayan sido explorados.

```
#include <iostream>
#include <queue>
#include <vector>
using namespace std;

typedef vector<vector<int> > List_Ad;

void bft(const List_Ad& t, int s, vector<int>& distance) {
  int n = t.size();
  vector<bool> examined(n,false);
  examined[s] = true;
  distance = vector<int>(n,-1); // distance[i] = -1 significa que i no es alcanzable
  distance[s] = 0;
  queue<int> not_exp;
  not_exp.push(s);
  while (not not_exp.empty()) {
    int act = not_exp.front();
    not_exp.pop();
    for (int i = 0; i < t[act].size(); ++i) {
      if (not examined[t[act][i]]) {
        examined[t[act][i]] = true;
        distance[t[act][i]] = distance[act] + 1;
        not_exp.push(t[act][i]);
      }
    }
    cout << "vertice " << act << " a distancia " << distance[act] << endl;
```

```
  }
}
```

El resultado de la llamada `bft(t,1,d)` para el grafo de la figura 3.1 es

```
vertice 1 a distancia 0
vertice 0 a distancia 1
vertice 5 a distancia 1
vertice 4 a distancia 2
vertice 2 a distancia 2
vertice 6 a distancia 2
vertice 3 a distancia 3
vertice 7 a distancia 3
```