Estructuras de Datos y Algoritmos 1

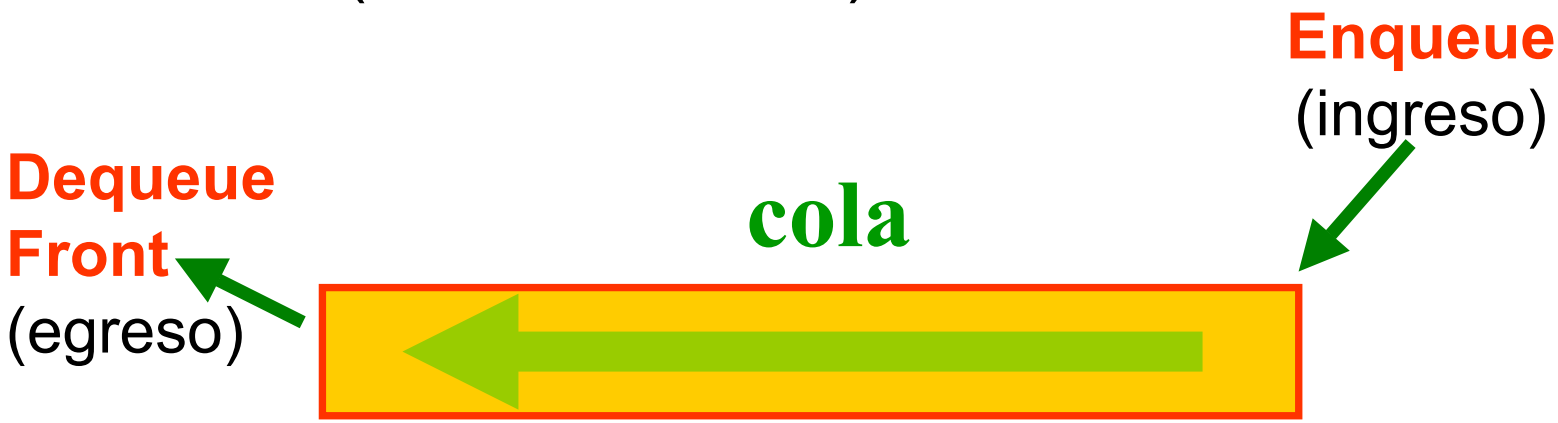
TAD COLA

Carlos Luna

Definición

Una ***Queue*** (cola) es otra clase especial de lista, donde los elementos son insertados en un extremo (el final de la cola) y borrados en el otro extremo (el frente de la cola).

Otro nombre usado para este tipo abstracto es el de lista FIFO (first-in-first-out).



Definición

Las operaciones para una cola son análogas a las de stack, la diferencia sustancial es que las inserciones son efectuadas al final de la lista.

La terminología tradicional para stacks y colas es también diferente.

Operaciones

El TAD Queue incluye las siguientes operaciones:

- la operación que construye una cola vacía, `Null`.
- una operación, `Enqueue`, que inserta un elemento al final de la cola.
- la operación `Front`, que retorna el elemento que se encuentra en el comienzo de la cola.
- la operación `Dequeue`, que borra el primer elemento de la cola.
- y finalmente, `IsEmpty`, que testea si la cola es vacía o no.

Módulo de definición para el TAD Queue

```
// módulo "Queue.h"
template <class Etype>
class Queue {
public:
  /* Constructoras */
  virtual void Null() = 0;
  // retorna la cola vacía

  virtual void Enqueue(const Etype &x) = 0;
  /* retorna una cola con último elemento x y
     el resto de la cola queda igual */
```

Definición de Queue (cont.)

```
  virtual bool IsEmpty() = 0;
  // retorna true si la cola es vacía
  virtual Etype Front() = 0;
  /* pre : la cola no es vacía
     post: retorna el primer elemento */
  virtual void Dequeue() = 0;
  /* pre : la cola no es vacía
     post: retorna la cola resultado de
           borrar el primer elemento */
};
```

Algunas Aplicaciones

- Prácticamente, toda fila real es (supuestamente) una cola: “se atiende al primero que llega”.
- Las listas de espera son en general colas (por ejemplo, de llamadas telefónicas en una central) .
- Los trabajos enviados a una impresora se manejan generalmente siguiendo una política de cola (suponiendo que los trabajos no son cancelados).
- Dado un servidor de archivos en una red de computadoras, los usuarios pueden obtener acceso a los archivos sobre la base de que el primero en llegar es el primero en ser atendido, así que la estructura es una cola (no siempre se usa este esquema).
- Existe toda una rama de las matemáticas, denominada *teoría de colas*, que se ocupa de hacer cálculos probabilistas de cuánto tiempo deben esperar los usuarios en una fila, cuán larga es la fila.....

Implementación del TAD Queue

- Al igual que para stacks, una implementación adecuada para este tipo abstracto es usar listas encadenadas.

- Sin embargo, en este caso se puede aprovechar el hecho de que todas las inserciones son efectuadas al final de la cola e implementar la operación **`Enqueue`** en forma eficiente. En vez de recorrer toda la lista cada vez que queremos insertar un nuevo elemento, lo que se hace es mantener un puntero al último elemento.

Implementación del TAD Queue (cont.)

- Al igual que para listas, también mantendremos un puntero (**`front`**) al comienzo de la misma.

  Para colas, este puntero permitirá implementar **`Front`** y **`Dequeue`** en forma eficiente.
- Se mantendrá además una celda "dummy" como cabeza de la lista, el puntero **`front`** apuntará a esta celda. Esta convención permitirá hacer un manejo adecuado de la cola vacía.
- A continuación definiremos un módulo de implementación para el tipo **`Queue`** basado en los conceptos descriptos arriba.

La clase QueueImp

```
// módulo QueueImp.cpp
#include "Queue.h"
#include <iostream.h>
#include <assert.h>
template <class Etype>
class QueueImp : public Queue<Etype>
{ protected:
     struct NodoQ {
          Etype elem;
          NodoQ *sig;};

     typedef NodoQ *PtrNodoQ;
     PtrNodoQ front,rear;
     void DeleteQueue();
```

La clase QueueImp

```
public:
  QueueImp();
  ~QueueImp() { DeleteQueue(); };
  void Null();
  void Enqueue(const Etype &x);
  bool IsEmpty();
  Etype Front();
  void Dequeue();
};
```

Implementación de operaciones

```
template <class Etype>
QueueImp<Etype>::
QueueImp()
{ front = new NodoQ;
  if (front == NULL)
     cout << "Out of space";
  else { front -> sig = NULL; rear = front; }
};

template <class Etype>
QueueImp<Etype>::
Null()
{ front -> sig = NULL;
  rear = front;
};
```

Enqueue e IsEmpty

```
template <class Etype>
void
QueueImp<Etype>::
Enqueue(const Etype &x)
{ rear -> sig = new NodoQ;
  rear -> sig -> elem = x;
  rear -> sig -> sig = NULL;
  rear = rear -> sig;
};

template <class Etype>
bool
QueueImp<Etype>::
IsEmpty() { return front == rear;};
```

Front y Dequeue

```
template <class Etype>
Etype
QueueImp<Etype>::
Front() { assert(front!=rear);
          return front -> sig -> elem; };

template <class Etype>
void
QueueImp<Etype>::
Dequeue()
{ assert(front!=rear);
  PtrNodoQ temp = front -> sig;
  front->sig = temp->sig;
  delete temp;};
```

DeleteQueue

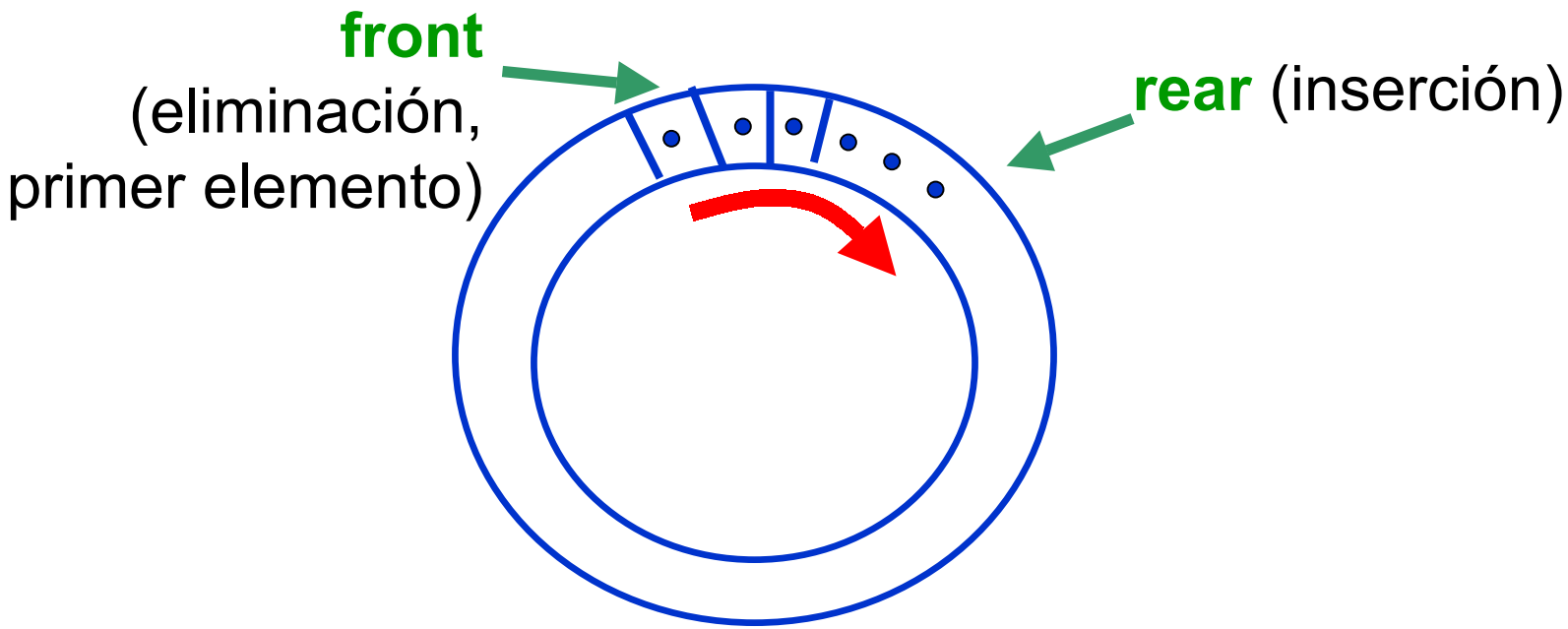
```
template <class Etype>
void
QueueImp<Etype>::
DeleteQueue(){
  while(!isEmpty()) Dequeue(); };
```

Versión acotada del TAD Queue e Implementación estática

- La representación que hemos visto para Stack haciendo uso de un **arreglo con tope** puede ser usada para implementar el tipo **Queue en su versión acotada** (en cantidad de elementos).
- Sin embargo, esta representación no es muy eficiente.
- La operación de agregar un elemento a la cola se puede ejecutar eficientemente: simplemente se incrementa el tope y en esa posición se da de alta al elemento.
- Pero dar de baja al primer elemento de la cola requiere desplazar todos los elementos una posición en el arreglo.

Implementación estática del TAD Queue

- Para solucionar este problema debemos tomar un enfoque distinto: un **arreglo circular**.
  Piense en un arreglo como un círculo, donde la primera posición del mismo "sigue" a la última.
- La cola se encontrará alrededor del círculo en posiciones consecutivas (en un sentido circular).



Implementación estática del TAD Queue

- Para insertar un elemento movemos el puntero al último una posición en el sentido de las agujas del reloj y en esa posición insertamos al elemento.
- Para dar de baja al primer elemento simplemente movemos al puntero primero una posición en el mismo sentido.
- De esta forma la ejecución de estas operaciones insume el mismo tiempo independientemente de la cantidad de elementos que componen a la estructura.

Implementación estática del TAD Queue

Esta representación tiene una desventaja:

es imposible distinguir una cola vacía de una que ocupa el arreglo entero.

Soluciones:

- Una variable entera que lleve el tamaño de la cola.
- Un indicador booleano que sólo se haga verdadero cuando la cola es vacía.
- Otra es prevenir que la cola no ocupe todo el arreglo.

Cuál le parece la solución más adecuada ?

Ejercicios Propuestos

- Implementar las operaciones de Queue con listas encadenadas pero sin considerar una celda auxiliar "dummy".
- Especificar e Implementar *Queue acotada* con arreglos “circulares”.

  Considerar una operación adicional isFull que retorna true si y sólo si la cola esta llena.

  Todas las operaciones de Queue acotada tienen la misma especificación que en Queue ?
- Desarrollar casos de prueba de las diferentes implementaciones vistas.

Bibliografía

- Estructuras de Datos y Análisis de Algoritmos en Pascal (o el de C++)

  Mark Allen Weiss; Benjamin/Cummings Inc., 1993.

⇒ **cap. 3 (3.4)**

O: Data Structures and Algorithm Analysis in C++

Mark Allen Weiss; Benjamin/Cummings Inc., 1994.

- Estructuras de Datos y Algoritmos

  A. Aho, J. E. Hopcroft & J. D. Ullman; Addison-Wesley, 1983.

⇒ **cap. 2 (2.4)**

- Como Programar en C/C++ (o la versión 2 del Deitel para C++)

  H.M. Deitel & P.J. Deitel; Prentice Hall, 1994.

Anexo 1: Especificación de Cola Acotada

```
/*////////////////////////////////////////////////
 Especifición de Cola Acotada (ColaEstatica.h)
////////////////////////////////////////////////*/
template <class T>
class ColaEstatica
{   public:
                        /*CONSTRUCTORAS*/
virtual void Empty()=0;
/*Vacía la cola*/
virtual void EnQueue(const T &x)=0;
/*pre:  Cola no llena
post: Inserta el elemento T al final cola*/
                        /*PREDICADOS*/
virtual bool isEmpty()=0;
/*Verifica si la cola es vacía o no*/
```

Anexo 1: Especificación de Cola Acotada

```
virtual bool isFull()=0;
/*Verifica si la cola está llena o no*/
                        /*SELECTORAS*/
virtual T Front()=0;
/*pre:  Cola no vacía
post: Retorna el elemento que esta el comienzo de la cola*/
virtual void DeQueue()=0;
/*pre: Cola no vacía
post: Elimina el primer elemento de la cola*/
                        /* EXTRA */
virtual void print()=0;
/*Imprime todos los elementos de la cola*/

};
```

Anexo 2: implementación de Cola Acotada con estructuras dinámicas

```
// ColaEstaticaCircularImp.cpp
#include <stdio.h>
#include <ostream.h>
#include <assert.h>
#include "ColaEstatica.h”
template<class T>
class ColaEstaticaCircularImp: public ColaEstatica<T>
{    protected:
            struct NodoCola { T info; NodoCola *sig; };
            typedef NodoCola *ptrNodoCola;
            ptrNodoCola front,rear;
            int CANTIDAD_NODOS;
            int contador;
     public:  //Constructor
            ColaEstaticaCircularImp<T>(int cant_nodos);
```

Anexo 2: implementación de Cola Acotada con estructuras dinámicas

```
//Destructor
    ~ColaEstaticaCircularImp();
    //Operaciones
    void Empty();
    void EnQueue(const T &x);
    bool isEmpty();
    bool isFull();
    T Front();
    void DeQueue();
    void print();
};
```

Anexo 2: implementación de Cola Acotada con estructuras dinámicas

```
template <classT>
ColaEstaticaCircularImp<T>::ColaEstaticaCircularImp(int cant_nodos)
{   front=NULL;
    if(cant_nodos>0) {
            rear=new NodoCola();
            front=rear;
            for (int k=0;k<cant_nodos-1;k++) {
                    rear->sig=new NodoCola();
                    rear=rear->sig; }
            rear->sig=front; rear=front;
            contador=0;
            CANTIDAD_NODOS=cant_nodos;
            };
            assert(front!=NULL);
};
```

Anexo 2: implementación de Cola Acotada con estructuras dinámicas

```
template <class T>
ColaEstaticaCircularImp<T>::~ColaEstaticaCircularImp()
{   for(int i=0; i<CANTIDAD_NODOS-1; i++)
            { ptrNodoCola aux=front; front=front->sig; delete aux; }
    delete front, rear;
};
template<class T>
bool ColaEstaticaCircularImp<T>::isEmpty()
{   return(contador==0); };
template<class T>
bool ColaEstaticaCircularImp<T>::isFull()
{   return(contador==CANTIDAD_NODOS); };
template<class T>
T ColaEstaticaCircularImp<T>::Front()
{   assert(!isEmpty()); return(front->info); };
```

Anexo 2: implementación de Cola Acotada con estructuras dinámicas

```
template <class T>
void ColaEstaticaCircularImp<T>::EnQueue(const T &x)
{   assert(!isFull());
    if(isEmpty()) rear->info=x;
    else { rear=rear->sig; rear->info=x; }
    contador++;
};
template <class T>
void ColaEstaticaCircularImp<T>::DeQueue()
{   assert(!isEmpty());
    contador--;
    if(contador==1) rear=rear->sig;
    front=front->sig; }
```

Anexo 2: implementación de Cola Acotada con estructuras dinámicas

```
template <class T>
void ColaEstaticaCircularImp<T>::Empty()
{   while(!isEmpty()) DeQueue(); }

template<class T>
void ColaEstaticaCircularImp<T>::print()
{   if(isEmpty()) cout<<"LA COLA ESTA VACIA"<<endl;
    else { ptrNodoCola aux=front;
           for(int i=0; i<contador; i++) {
           cout<< "Elemento " << i+1 << ": "<< aux->info <<endl;
                  aux=aux->sig; }
    cout<<endl;  }
}
```

Anexo 3: casos de prueba

```
#include "ColaEstaticaCircularImp.cpp”
void prueba();
void verificarColaVacia(ColaEstatica<int> *c);
void retornarPrimerElemento(ColaEstatica<int> *c);
void verificarColaVacia(ColaEstatica<int> *c)
{   if(c->isEmpty()) cout<< "COLA VACIA"<<endl;
    else cout<< "COLA NO VACIA"<<endl; }
void retornarPrimerElemento(ColaEstatica<int> *c)
{   if(c->Front()==0) cout<<"COLA VACIA"<<endl;
    else cout<<"Primer Elemento: "<< c->Front()<<endl; }
void prueba() {
ColaEstatica<int> *c= new ColaEstaticaCircularImp<int>(10);
// verificarColaVacia(c);
// retornarPrimerElemento(c);
```

Anexo 3: casos de prueba

```
c->print(); c->EnQueue(1); c->EnQueue(2); c->EnQueue(3);
c->EnQueue(4);c->EnQueue(20); c->EnQueue(6); c->print();
c->EnQueue(7);retornarPrimerElemento(c); c->DeQueue();
retornarPrimerElemento(c); c->Empty(); verificarColaVacia(c);
c->EnQueue(8);retornarPrimerElemento(c); c->EnQueue(9);
c->print();
}

void main()
{   prueba(); }
```