Ciencias computacionales

Propedéutico: Programación

Dra. Hayde Peregrina

Contents

1 Pilas y Colas

Video sobre pilas.
Video sobre colas.

1.1 Introducción

Al contrario de las estructuras estáticas como lo son las listas y los arreglos, donde el tamaño asignado no puede cambiar durante el tiempo de ejecución. Existen estructuras que pueden cambiar dicho tamaño durante la ejecución de un programa como lo son las pilas y las colas. Una pila sigue lo que se conoce como último en entrar primero en salir (LIFO, Last In First Out). Por ejemplo, haciendo alusión a una "pila de platos", cuando un plato se necesita se toma del tope o de la cima. Cuando se agrega un plato se pone en el tope de la pila. Notemos que si en este momento se necesita un plato, se toma el plato más "nuevo" de la pila. Es decir, la pila exhibe la propiedad LIFO [1, 2].

Una pila o *stack* se define como:

- Una colección de datos que sólo puede ser accedida por un extremo el cual comúnmente se conoce como *tope*.

Las pilas no son estructuras fundamentales de datos (pueden ser representadas como *datos abstractos*), es decir, no están definidas como tales en los lenguajes de programación pero pueden representarse mediante: *arreglos* y *listas enlazadas*.

Por otro lado las, el tipo de dato abstracto conocido como *cola* inserta sus elementos por un extremo y los atiende por el extremo contrario, a esto se le conoce como primero en entrar primero en salir (FIFO, First In First Out) [1, 2]. Los ejemplos más representativos para describir a las colas son: las colas de los bancos, en los supermercados en un concierto o en algún otro evento deportivo.

En el computo son las tareas que están en espera a ser ejecutadas o cuando se desea realizar un trabajo por varios usuarios a la vez, un claro ejemplo es cuando varios usuarios mandar a imprimir algún documento a una impresora en red. Ya que la impresora unicamente puede manejar un trabajo a la vez, los otros tendrán que ser encolados.

1.2 Operaciones y Representación

Las operaciones más comunes para las pilas son: push y pop. Se entiende como *push* a la acción de poner o meter elementos a la pila y se por *pop* se entiende como quitar o sacar elementos de la pila. La Fig. 1, muestra las operaciones push y pop respectivamente.

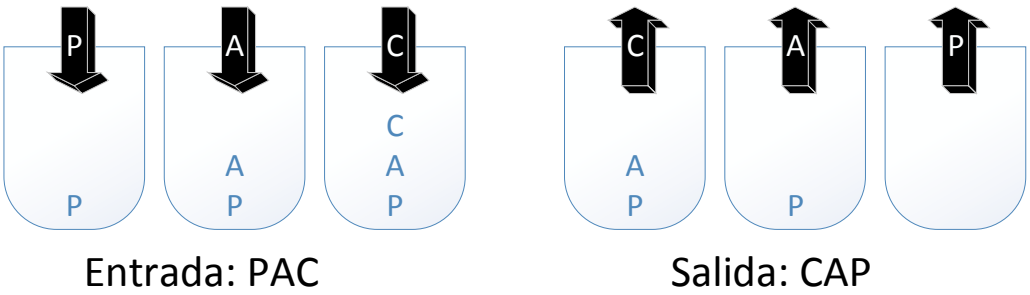

Figure 1: Operaciones push y pop en una pila.

Otras operaciones básicas contempladas a la hora de implementar una pila son: es *vacía* (cuando la pila no contiene elementos) y esta *llena* (cuando se llega al número máximo de elementos, es decir, no caben más elementos). Estas operaciones sirven para detectar errores como de desbordamiento. *Underflow* cuando se intenta extraer un valor de una pila vacía. Y *overflow* cuando se intenta meter un elemento en una pila llena.

Por otro las operaciones para las colas son: encolar (queue) y desencolar (dequeue). Se supone que se conoce el frente de la cola (por donde se desencolan los elementos) y también la parte trasera o final de la cola (por donde se encolan los elementos). Es decir, el primer elemento que entra en la cola será atendido y será el primero en salir, respetando un orden FIFO. La Fig. 2, muestra las operaciones queue y dequeue respectivamente.

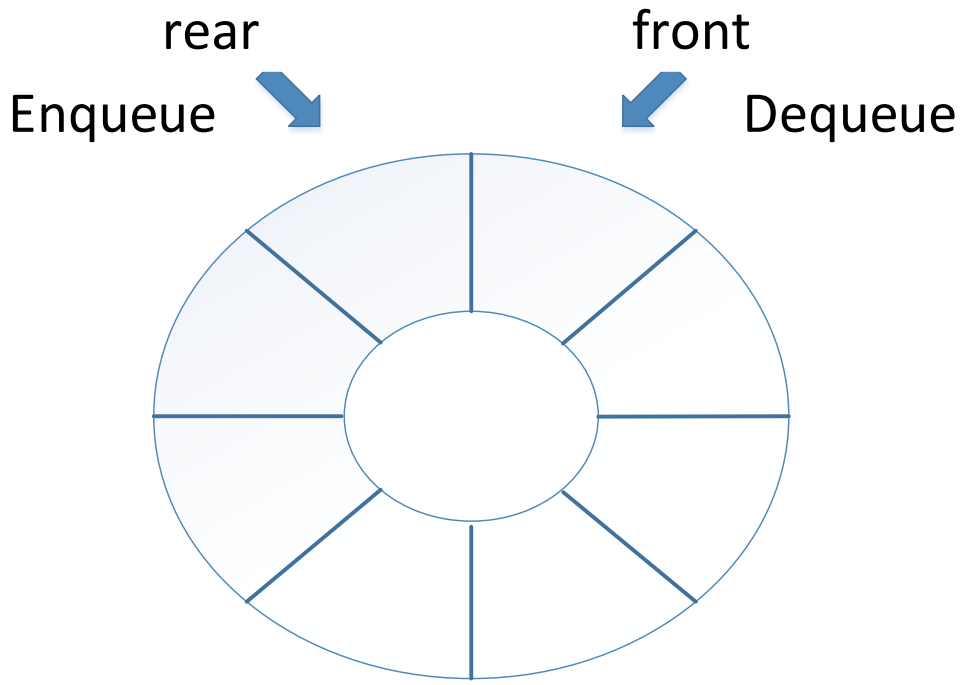

Figure 2: Operaciones queue y dequeue en una cola circular.

Implementación de pilas usando arreglos

Las pilas pueden ser representadas de distintas maneras las más comunes son: mediante *arreglos* y utilizando *listas ligadas*. La diferencia es debido a que se hace de manera estática cuando se usan los arreglos y de manera dinámica utilizando listas ligadas [3].

Para su implementación mediante arreglos se necesita conocer o definir el índice tope de la pila y el número máximo de elementos. A la hora de iniciar se define la pila como vacía en la posición -1 sin ningún elemento, de esta forma el primer elemento queda en la posición 0, el segundo en la posición 1 y así sucesivamente [1].

De lo anterior se pueden definir los algoritmos sobre datos para hacer push y pop en una pila.

Push (Insertar)

- Revisar si la pila no está llena.
- Incrementar el indice más 1.
- Almacenar el elemento en la posición del apuntador.

Pop (Sacar o quitar)

- Revisar si la pila no está vacía.
- Leer o extraer el elemento del apuntador de la pila.

- Disminuir el indice menos 1 de la pila.

Se puede declarar un *archivo.h* que contenga a la pila y sus tipos de datos asociados a ella, así como los prototipos de sus funciones, como se muestra a continuación:

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE_STACK 100

typedef int dataType ; /* Data type for the elements of the stack */

typedef struct{
        dataType arrayStack [SIZE_STACK] ;
        int top;
        int *s;
}Stack;

/*Stack operations*/
void push (Stack *stack, dataType element) ;
dataType pop (Stack *stack) ;
int isEmpty (Stack stack) ;
int isFull (Stack stack) ;
```

La operación push o insertar un elemento cualquiera incrementa el tope en uno, y asigna el nuevo elemento al arreglo de la pila. Esta función estaría declarada en el programa principal.

```
#include "stack.h"
typedef int dataType;

void push (Stack *stack, dataType element){
        if (isFull(*stack)){
                puts ("Stack␣Overflow") ;
                exit (1) ;
        }
        stack -> top++ ;
        stack -> arrayStack[stack->top] = element ;
}
```

La operación pop o quitar extrae un elemento de la pila. Primero se copia dicho elemento en una variable auxiliar y se procede a disminuir el apuntador del tope menos uno. Se debe considerar el caso en el cual la pila esta vacía y devolver un mensaje de error.

```
#include "stack.h"
typedef int dataType;

dataType pop (Stack *stack){
        dataType temp;
        if (isEmpty(*stack)){
                puts ("Stack␣Overflow") ;
                exit (1) ;
        }

        temp = stack -> arrayStack[stack->top] ;
        stack -> top -- ;
        return temp ;
}
```

Implementación de pilas usando listas ligadas

Las pilas también pueden ser implementadas de manera dinámica haciendo uso de las listas ligadas. Sólo se tiene que tener en cuenta que las pilas crecen reservando memoria, es decir, es necesario liberar memoria según se extraigan elementos de dicha pila [2, 3].
La implementación basada en listas ligadas también implementa las operaciones push y pop, siendo la inserción y extracción de un elemento por un mismo extremo. Se cuenta con las mismas operaciones que para las pilas basadas en arreglos a excepción de *isFull*, ya que la única restricción es el límite de memoria, la pila podría crecer de manera infinita, la Fig. 3 muestra un ejemplo de pila usando listas ligadas.

Figure 3: Implementación de una pila basada en lista ligada.

De igual manera se puede declarar un *archivo.h*, llamado *stack.h* por ejemplo, que contenga a la pila y sus tipos de datos asociados a ella, así como los prototipos de sus funciones, como se muestra a continuación:

```
1  #include <stdlib.h>
2
3  typedef struct{
4          dataType element ;
5          struct node *next;
6  }Node;
7
8  /*Stack operations*/
9  void createStack (Node** stack) ;
10 void push (Node **stack, dataType element) ;
11 dataType pop (Node **stack) ;
12 int emptyStack (Node *stack) ;
13 dataType top (Node *stack) ;
```

A continuación se muestra un ejemplo, desde la creación de la pila y sus principales operaciones asociadas a ella.

```
1  #include "stack.h"
2
3  typedef int dataType ;  /* Data type for the elements of the stack*/
4
5  void createStack (Node **stack){
6          *stack = NULL;
7  }
8
9  int emptyStack (Node *stack){
10         return pila == NULL ;
11 }
```

Ahora vamos a insertar o hacer *push* sobre la pila. Para esto es necesario crear un nodo y asignar una parte de memoria ademas de actualizar el tope de la pila.

```
1  void push (Node **stack, dataType element){
2          Node * new ;
3          new = (Node*)malloc (sizeof(Node)) ; // Create a new node
```

```
4           new -> element = element ;
5           new -> next = *stack ;
6           (*stack) = new ;
7   }
```

Ahora veamos el caso en el cual se desea extraer o leer el un elemento de la pila sin hacerle pop a dicho elemento. También se muestra el caso pop.

```
1   dataType top (Node *stack){
2           if (emptyStack (stack)){
3                   puts ("Error, empty stack") ;
4                   exit (1) ;
5           }
6           return stack -> element ;
7   }
```

En el caso de pop se libera el espacio de memoria que aloja al último elemento insertado dentro de la pila, es decir libera el tope y se devuelve dicho elemento.

```
1   dataType pop (Node *stack){
2           dataType temp;
3           Node * n ;
4
5           if (emptyStack (*stack)){
6                   puts ("Error, empty stack") ;
7                   exit (1) ;
8           }
9           temp = (*stack) -> element ;
10          n = *stack ;
11          (*stack) = (*stack) -> next ;
12          free (n) ;
13
14          return temp ;
15  }
```

Implementación de colas usando arreglos

Al contrario de las pilas en el tipo de dato abstracto conocido como *cola* se atiende al primero en llegar, es decir, se asemejan o derivan de una cola de personas en una taquilla. Las colas pueden ser implementadas ya sea usando arreglos o mediante el uso de listas ligadas [1, 3].
Las colas atienden en el orden en que entran los elementos, es decir, el primero que entra será el primero en ser atendido y por ende en salir de la cola. A esto se le conoce como (FIFO, first-in first-out). Las principales operaciones asociadas a las colas son: *enqueue, queue* o encolar (agregar elementos), *dequeue* quitar o extraer elementos, si es basada en arreglos la operación *isFull* verifica si la cola esta llena. Dicho problema es mitigado cuando la implementación de la cola es basada en listas ligadas. Por último también se puede verificar si la cola está vacía mediante la operación *isEmpty*.

Las colas basadas en arreglos deben mantener un puntero al frente (*front*) y otro al fin o final (*rear*) de la cola. Al agregar (*enqueue o queue*) un elemento se debe verificar que la posición dentro de la cola sea valida y después incrementar el apuntador *final + 1*. En caso contrario se al quitar (*dequeue*) se debe verificar que la cola no sea vacía para posteriormente poder hacer un decremento del apuntador *frente*.

La Fig. 4 muestra los inconvenientes o desventajas de implementar una cola de forma lineal basada en arreglos. Es decir, el apuntador *frente* pudiera alcanzar al apuntador *final* sin que se puedan insertar más elementos, pero habría posiciones libres a la izquierda del apuntador *frente*.

Para mitigar o resolver las desventajas anteriormente mencionadas, se considera una implementación de colas basadas en un arreglo circular.

Figure 4: Cola lineal basada en arreglos.

Implementación de colas circulares con arreglos

Las colas circulares al contrario de las colas lineales utilizan todos los espacios, es decir, no dejan posiciones libres a las que no se pueda tener acceso. Al igual que la cola lineal la cola circular también necesita dos apuntadores (*frente y final*). El apuntador *frente* queda fijo y el que se mueve es el apuntador *final*, ya que por este extremo se asigna un nuevo elemento a la cola. La Fig. 5 muestra una cola vacía y una cola con un elemento, ejemplificando como se deben mover los apuntadores *frente* y *final*.

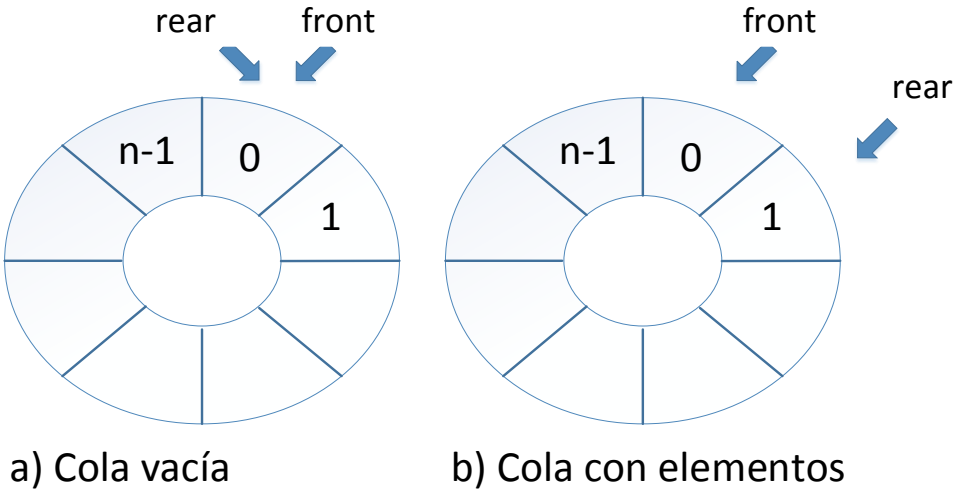

Figure 5: Cola circular basada en arreglos.

El movimiento circular de los índices se hace tomando al final de la cola como: $final = (final+1)\%n$. Por ejemplo para una cola de tamaño 5 se tendría $final = (4+1)\%5 = 0$. Para un elemento $final =$

$(0+1)\%5 = 1$ y así sucesivamente.

A continuación se muestra un ejemplo, desde la creación de la cola y sus principales operaciones asociadas a ella.

```
frente = 0 ;
final = MAX_SIZE - 1 ;

void createQueue (Queue *queue){
        queue -> front = 0 ;
        queue -> rear = MAX_SIZE - 1 ;
}
```

Para insertar o encolar un elemento se debe avanzar de manera circular su apuntador asociado a final o *rear*.

```

void enQueue (Queue *queue, dataType element){
        if (isFull (*queue)){
                puts ("Queue Overflow") ;
                exit (1) ;
        }
        queue -> rear = next (queue->rear) ;
        queue -> arrayQueue[queue -> rear] = element ;
}
```

Para quitar o desencolar el índice frente debe avanzar circularmente.

```
dataType deQueue (Queue *queue){
        dataType temp;
        if (isEmpty (*queue)){
                puts ("Queue Overflow") ;
                exit (1) ;
        }
        temp = queue -> arrayQueue[queue -> front] ;
        queue -> front = next (queue->front) ;
        return temp ;
}
```

A continuación se muestra un ejemplo de como codificar si la cola está vacía o esta llena.

```

int isEmpty (Queue queue){
        return queue.front == next (queue.rear) ;
}

int isFull (Queue queue){
        return queue.front == next (next(queue.rear)) ;
}
```

Implementación de colas usando listas ligadas

Existen varios inconvenientes a la hora de implementar colas utilizando arreglos: por ejemplo, desperdicio de memoria o no poder almacenar más elementos. Una alternativa atractiva sería implementar de manera dinámica la cola, esto se logra mediante ligas ligadas. Aunque se desperdicia memoria para mantener los enlaces, sigue siendo una solución eficaz.

Esta implementación se ajusta de acuerdo al número de elementos que se almacenan en cola. Es necesario mantener dos apuntadores que son los encargados de encolar (apuntador final) y de extraer (apuntador frente).

A continuación se muestra un ejemplo de archivo.h (*llamado cola.h*) donde se pueden declarar los tipos de datos y los prototipos de las funciones básicas de una cola implementada de manera dinámica.

```
1
2  #include <stdlib.h>
3
4  struct node{
5          dataType element;
6          struct node* next;
7  };
8  typedef struct node Node;
9
10 typedef struct{
11         Node* front;
12         Node* rear;
13 }Queue;
14
15 void crearQueue (Queue* queue);
16 void enQueue (Queue* queue, dataType element);
17 dataType deQueue (Queue* queue);
18
19 dataType front (Queue queue);
20 int emptyQueue (Queue queue);
```

A continuación se muestran las funciones anteriores utilizadas dentro de un programa principal.

```
1  #include "cola.h"
2  typedef char* dataType;
3
4  void createQueue (Queue* queue){
5          queue -> front = queue -> rear = NULL ;
6  }
7
8  Nodo* createNode (dataType element){
9          Node* n ;
10         n = (Node*) malloc (sizeof(Node)) ;
11         n -> element = element ;
12         n -> next = NULL ;
13         return n;
14 }
15
16 int emptyQueue (Queue queue){
17         return (queue.front == NULL) ;
18 }
19
20 void enqueue (Queue* queue, dataType element)
21 {
22         Nodo* a;
23         a = crearNodo (element);
24         if (emptyQueue (*queue) ){
25                 queue -> front = a ;
26         }
27         else{
28                 queue -> rear -> next = a ;
29         }
30         queue -> rear = a ;
31 }
```

```
32
33 dataType dequeue (Queue* queue){
34
35         dataType temp ;
36         if (!emptyQueue (*queue)){
37                 Nodo* n ;
38                 n = queue -> front ;
39                 temp = queue -> front -> element ;
40                 queue -> front = queue -> front -> next ;
41                 free (n) ;
42         }
43         else {
44                 puts ("Overflow␣Empty␣Queue");
45                 exit (-1);
46         }
47
48         return temp;
49 }
50
51 dataType front (Queue queue){
52         if (emptyQueue(queue)){
53                 puts ("Overflow␣Empty␣Queue␣") ;
54                 exit (-1);
55         }
56         return (queue.front -> element) ;
57 }
```

1.3 Aplicaciones

Aplicaciones de las pilas

Las aplicaciones comunes de las pilas son: sistemas operativos, compiladores, navegadores Web, editores de texto y programas de aplicaciones. Otras aplicaciones comunes para las pilas se pueden encontrar a la hora de evaluar expresiones aritméticas. Las pilas también son usadas por JVM (Java Virtual Machine).

Aplicaciones de las colas

Varios trabajos dentro del mundo de la computación son delegados a las colas los ejemplos típicos son colas de mensajes, colas de prioridades y como se atienden a los usuarios cuando se manda a imprimir algún documento por varios usuarios a la vez. También se puede decir que donde exista concurrencia se utilizaran las colas para almacenar elementos. Las colas también son consideradas o utilizadas cuando se hacen solicitudes a un servidor.

References

[1] Noel Kalicharan. *Advanced Topics in C: Core Concepts in Data Structures*. Apress, Berkely, CA, USA, 1st edition, 2013.

[2] Alfred V Aho, John E Hopcroft, Jeffrey D Ullman, Américo Vargas Villazón, and Jorge Lozano Moreno. *Estructuras de datos y algoritmos*, volume 1. Addison-Wesley Iberoamericana, 1988.

[3] Nell Dale. *C++ plus data structures*. Jones & Bartlett Learning, 2007.