Pilas y Colas

Apuntes para la materia de Estructuras de Datos y Algoritmos I

Facultad de Ingeniería, UNAM

Marzo-17

(Rev0.2)

Este es un trabajo en progreso.

Índice

1 INTRODUCCIÓN

Las pilas y colas son estructuras de datos ampliamente utilizadas en la programación de software porque modelan comportamientos del mundo real y surgen naturalmente en actividades de resolución de problemas con la computadora.

Ambas, pilas y colas, pueden ser codificadas con una amplia variedad de lenguajes e implementaciones. En este documento nos concentraremos en su implementación con arreglos tipo C y con listas enlazadas, y serán modeladas como TDUs.

Prerequisitos:

- Dominio de los elementos básicos del lenguaje C
- Estructuras
- Arreglos y apuntadores
- ADTs en C
- Listas enlazadas
- Uso de la macro assert()

2 PILAS

2.1 Definición

Una pila es una estructura de datos que sólo soporta inserciones y extracciones por un solo lado, y debido a este comportamiento también se le conoce como una estructura LIFO (Last in, first out) porque el último elemento en ser insertado es el primero que puede ser extraído. Si se desea accesar a un elemento que no es el primero, entonces se necesita ir sacando (y quizás descartando) los elementos que están por encima de él.

gráfica

2.2 Operaciones

Desde el punto de vista de un cliente de una pila se tienen las siguientes operaciones. Cabe destacar que éstas son independientes de la implementación (por ejemplo, si la pila está codificada con arreglos o con listas enlazadas):

- **`Create`**. Crea una nueva pila.
- **`Push`**. Inserta un elemento en la parte superior de la pila.
- **`Pop`**. Extrae y devuelve el elemento en la parte superior de la pila.
- **`Peek`**. Permite ver el valor del elemento en la parte superior de la pila, pero sin sacarlo (la pila no se altera).
- **`IsEmpty`**. Indica si la pila está vacía.
- **`IsFull`**. Indica si la pila está llena (sólo tiene sentido con las pilas basadas en arreglos).
- **`Clear`**. Borra los elementos de la pila (pero no la destruye). Piense en esta operación como una reinicialización de la pila

2.2.1 Cómo funciona

Empecemos mencionando que cuando se habla de pilas surge el término "parte superior de la pila", o "tope de la pila". Por convención nos referiremos a este término como el "*top*" de la pila.

Originalmente la pila está vacía y *top* apunta a la primera localidad válida de la pila. Cuando se hace un *Push* el elemento se inserta en donde apuntaba *top*, y *top* se incrementa en uno para apuntar a la siguiente localidad por encima de la actual y quedar lista para la siguiente operación *Push*.

gráfica

Este procedimiento se repite con cada *Push*. Cuando se hace un *Pop* se decrementa *top* y se extrae el elemento apuntado por *top*. El decremento de *top* es necesario porque la operación *Push* hace que *top* quede por encima del último elemento.

Casos especiales

La implementación particular de la pila debe hacerse cargo de los siguientes casos especiales:

- La pila no existe.
- Se pretende hacer un *Push* con una pila llena (en caso de pilas implementadas con arreglos).
- Se pretende hacer un *Pop* con una lista vacía.
- Se pretende hacer un *Peek* con una lista vacía.
- Se pretende destruir una pila que no existe o que fue destruída anteriormente.

2.3 Implementación con arreglos

La implementación de una pila con arreglos se puede hacer con arreglos estáticos o con arreglos dinámicos. Veremos las dos implementaciones.

2.3.1 Pilas con arreglos estáticos, versión canónica

Esta versión de una pila la escribiremos sin usar funciones para que el alumno observe la parte interna del funcionamiento. La siguiente versión ya usará funciones.

```
#include <assert.h>
// para usar las aserciones

#define MAX_TAM 5
int main()
{
    double pila[MAX_TAM];
    int top = 0;

    if (top >= MAX_TAM) { assert(0); }
    pila[top] = 2.5;
    ++top;

    if (top >= MAX_TAM) { assert(0); }
    pila[top] = 3.5;
    ++top;
```

```
    if (top >= MAX_TAM) { assert(0); }
    pila[top] = 4.5;
    ++top;


    if (top == 0) { assert(0); }
    --top;
    printf ("<%f>\n", pila[top]);

    if (top == 0) { assert(0); }
    --top;
    printf ("<%f>\n", pila[top]);

    if (top == 0) { assert(0); }
    --top;
    printf ("<%f>\n", pila[top]);

    return 0;
}
```

El lector podrá observar tres elementos importantes de una pila:

- **TAM_MAX**. Es el máximo número de elementos que la pila puede tener
- **double pila[TAM_MAX]**. Es el arreglo que implementa a la pila. En este ejemplo el arreglo es de tipo **double**, pero puede ser cualquier tipo válido en C.
- **top**. Es el apuntador al elemento en la parte superior de la pila (por convención se le llama apuntador, pero en esta implementación es un entero que se usa como índice del arreglo).

Así mismo, quedan claramente expuestas las instrucciones para hacer un *Push* y un *Pop* sobre la pila.

2.3.2 Pilas con arreglos estáticos, versión con funciones

Esta versión de una pila es similar a la anterior, pero ahora usa funciones para las operaciones de *Push* y *Pop*.

```
#define MAX_TAM 3

void push (double * stack, int * top, double newVal)
{
	if (*top >= MAX_TAM) { assert (0); }

	stack[*top] = newVal;
	++(*top);
}

double pop (double * stack, int * top)
{
	if (*top == 0) { assert (0);  }

	--(*top);

	return stack[*top];
}

int main()
{
	double pila[MAX_TAM];
	int top = 0;

	push (pila, &top, 2.5);
	push (pila, &top, 3.5);
	push (pila, &top, 4.5);

	printf ("<%f>\n", pop (pila, &top));
	printf ("<%f>\n", pop (pila, &top));
	printf ("<%f>\n", pop (pila, &top));

	return 0;
}
```

Vale la pena notar que para las operaciones Push y Pop, el primer argumento es un arreglo, mientras que el segundo argumento es el apuntador al *top* de la pila. Esta configuración nos va a permitir tener varias pilas en el mismo programa, y al mismo tiempo evitamos el uso de variables globales (como es común en algunos textos sobre este tema). Hay que mencionar que esta forma de programar nos está llevando al objetivo final que es tener un ADT para una pila.

Nota: El primer argumento de *Push* y *Pop* es un arreglo. Cuando se realiza una llamada a cualquiera de estas funciones sólo basta con escribir el nombre del arreglo (sin ampersand) ya que el nombre del arreglo es su dirección de inicio. El segundo argumento es el apuntador (en este caso un entero) al elemento superior de la pila. Las operaciones *Push* y *Pop* van a cambiar el valor de la variable **`top`**, por lo que necesitamos mandarla como referencia (sí requiere el ampersand).

2.3.3 Pilas con arreglos estáticos, versión ADT

Esta versión es la más interesante porque estaremos implementando un ADT para una pila, y de esta forma el futuro cliente del ADT sólo se concentrará en usar a la pila, sin tener que pensar en los detalles internos del mismo. El lector deberá notar que el arreglo contenedor se le está *inyectando* al nuevo objeto Stack; esto es, el arreglo NO es parte del ADT, sino que fue declarado fuera de él, y luego es usado por el ADT. Esta forma de trabajo es conveniente cuando no se desea –o no se puede– utilizar memoria dinámica; sin embargo, más adelante el arreglo deberá ser parte del ADT utilizando la función **malloc()** cuando se cree cada objeto tipo **Stack**. Este será un ejercicio de la práctica correspondiente a este tema.

```
typedef enum { FALSE = 0, TRUE = !FALSE } Bool;

struct Stack_Type
{
	double * stack;
	int top;
	int max;
};

typedef struct Stack_Type Stack;

void Stack_Ctor (Stack * this, double * stack, int max)
{
	this->stack = stack;
	this->max = max;
	this->top = 0;
}

void Stack_Push (Stack * this, double newVal)
{
	if (this->top >= this->max) { assert (0); }

	this->stack[this->top] = newVal;

	++this->top;
}

double Stack_Pop (Stack * this)
{
	if (this->top == 0) { assert (0); }

	--this->top;

	return this->stack[this->top];
}
```

```
Bool Stack_IsEmpty (Stack * this)
{
	return (this->top == 0);
}

Bool Stack_IsFull (Stack * this)
{
	return (this->top >= this->max);
}


#define MAX_TAM 3

int main()
{
	Stack miPila;
	double arreglo[MAX_TAM];
	Stack_Ctor (&miPila, arreglo, MAX_TAM);

	Stack_Push (&miPila, 2.5);
	Stack_Push (&miPila, 3.5);
	Stack_Push (&miPila, 4.5);

	printf ("<%f>\n", Stack_Pop (&miPila));
	printf ("<%f>\n", Stack_Pop (&miPila));
	printf ("<%f>\n", Stack_Pop (&miPila));

	Stack otraPila;
	double arreglo2[100];
	Stack_Ctor (&otraPila, arreglo2, 100);

	// como los contenedores son arreglos locales a esta función, entonces NO es necesario liberar ninguna memoria.

	return 0;
}
```

En el *driver program* (es decir, la función **main()**) se puede ver que con el mismo código podemos tener más de una pila para el mismo programa. Y como en nuestros ADTs anteriores, el primer argumento es una referencia a un objeto del mismo tipo que el ADT, en este ejemplo, un objeto tipo **Stack**.

Usando las funciones Is_Full() y Is_Empty()

El siguiente *driver program* nos muestra una forma de usar a dichas funciones.

```
#define MAX_TAM 5

int main()
{
	Stack miPila;
	double arreglo[MAX_TAM];

	Stack_Ctor (&miPila, arreglo, MAX_TAM);

	while (Stack_IsFull (&miPila) == FALSE) {
		double valor = 0.0;

		printf ("Nuevo valor: ");
		scanf ("%lf", &valor);

		Stack_Push (&miPila, valor);
	}

	while (Stack_IsEmpty (&miPila) == FALSE) {
		printf ("<%f>\n", Stack_Pop (&miPila));
	}

	return 0;
}
```

2.3.4 Pilas con arreglos dinámicos, versión ADT

Esta última versión de pilas con arreglos le delegará al ADT la tarea de crear:

- Un objeto dinámico tipo **Stack**; es decir, ahora dicho objeto estará en el área de memoria *heap*.
- El arreglo para la pila también estará en el *heap*.

Así, el cliente del ADT sólo tendrá que establecer el tamaño máximo de la pila, y el ADT se encargará del resto. Sin embargo, ahora hay que incluir una función para devolver la memoria asignada en forma dinámica, y cambió la firma y el código de la función constructora para poder crear los objetos y arreglos en el *heap*.

El código es prácticamente el mismo, así que sólo se mostrarán los nuevos cambios y el *driver program* para que el usuario vea la forma de utilizarlo.

```
// … igual que antes

Stack * Stack_Ctor (int max)
{
	Stack * newStack = (Stack *) malloc (sizeof (Stack));
	// crea a un nuevo objeto Stack en el heap

	assert (newStack);

	newStack->stack = (double *) malloc (max * sizeof (double));
	// crea a un nuevo contenedor (el arreglo) de double's en el heap

	assert (newStack->stack);

	newStack->top = 0;
	newStack->max = max;

	return newStack;
}

void Stack_Dtor (Stack * this)
{
	assert (this);

	free (this->stack);
	// devuelve la memoria del arreglo

	free (this);
	// devuelve la memoria del objeto
}

// … igual que antes

#define MAX_TAM 5

int main()
{
	Stack * miPila = Stack_Ctor (MAX_TAM);

	while (Stack_IsFull (miPila) == FALSE) {
		double valor = 0.0;

		printf ("Nuevo valor: ");
		scanf ("%lf", &valor);

		Stack_Push (miPila, valor);
	}
```

```
    while (Stack_IsEmpty (miPila) == FALSE) {
        printf ("<%f>\n", Stack_Pop (miPila));
    }

    Stack_Dtor (miPila);
    // en esta versión ¡no olvidar llamar al destructor!

    return 0;
}
```

Nota: El lector notará que el objeto **miPila** no está precedido por el ampersand (es decir, el operador referencia) en las llamadas a las funciones del ADT. Esto es porque en esta versión **miPila** fue declarado como un puntero. La función **Stack_Ctor()** devuelve la dirección de un objeto **Stack** en el heap y ésta se asigna al (ahora) apuntador **miPila**. Todas las funciones del ADT (excepto **Stack_Ctor()**) requieren que el primer argumento sea una dirección, y como **miPila** es efectivamente una dirección es así que podemos omitir al operador referencia (el ampersand).

2.3.5 Ejercicios

1 Escriba un programa que lea una palabra y la imprima al revés utilizando una pila de caracteres.
2 Escriba un programa que implemente una calculadora pre-fix.
3 Escriba un programa que convierta una expresión in-fix en pre-fix.
4 Investigue lo que son las expresiones pre-fix, in-fix y post-fix.

3 COLAS

Una cola es una estructura de datos que soporta inserciones y extracciones por ambos lados del contenedor, pero la restricción es que se inserta por un extremo y se extrae por el otro. Debido a esto también se les conoce como listas FIFO (first in, first out), es decir, el primer elemento en entrar será el primer elemento en salir, y por supuesto, el último en entrar será el último en salir. Las colas se utilizan para modelar cualquier contexto en el que el órden de llegada de los elementos deba ser respetado cuando se extraen. Lo primero que nos viene a la mente son las "filas" que tenemos que hacer en el banco, las tortillas, la biblioteca, etc. Otro uso muy extendido de las colas son los búfers. Éstos son áreas de memoria que tienen que almacenar una secuencia de elementos antes de ser extraídos y procesados. Sin ir muy lejos, un ejemplo común es el teclado de la computadora que almacena los caracteres escritos en un búfer y los libera (en el mismo órden en el que entraron) cuando el usuario presiona la tecla Enter, o el búfer se llena. También los archivos usan un búfer antes de ser escritos en el disco duro.

Existen algunas variantes de este tipo de estructuras. Las recién mencionadas son las colas lineales (o simplemente colas), mientras que existen también las colas circulares, las cuales cuando llegan al final del contenedor "dan la vuelta" al principio de ésta para seguir insertando o extrayendo.

3.1 Operaciones

Desde el punto de vista de un cliente de la pila se tienen las siguientes operaciones. Cabe destacar que éstas son independientes de la implementación (por ejemplo, si la cola está codificada con arreglos o con listas enlazadas) y del tipo de la misma (cola lineal o cola circular):

- **`Create`**. Crea una nueva pila.
- **`Insert`**. Inserta un elemento al final de la cola .
- **`Remove`**. Extrae y devuelve el elemento al principio de la cola.
- **`IsEmpty`**. Indica si la cola está vacía.
- **`IsFull`**. Indica si la cola está llena (sólo tiene sentido con las pilas basadas en arreglos).
- **`Clear`**. Borra los elementos de la cola (pero no la destruye). Piense en esta operación como una reinicialización.

3.1.1 Cómo funciona

Todas las colas requieren dos apuntadores, uno a la localidad de memoria más vieja en el tiempo (es decir, en la que se guardó el primer elemento) y a la cual le vamos a llamar *first*, y otro apuntador a la localidad más nueva en el tiempo (es decir, en la que se guardó el último elemento en llegar) y a la cual le vamos a llamar *last*. Cuando la cola está vacía ambos *first* y *last* apuntan al primer elemento del contenedor. Cuando se hace una inserción el elemento es colocado a donde apunta *last*, y éste se mueve una posición hacia adelante, para quedar listo para la siguiente inserción. Cuando se hace una extracción se devuelve el elemento apuntado por *first*, y éste se mueve una posición hacia adelante, para quedar listo para una nueva extracción. Tanto para las extracciones como para las inserciones es necesario preguntar si la lista está vacía o no, o si está llena o no, respectivamente. Esto lo veremos más adelante.

Problemas con las colas lineales. Un problema inherente de este tipo de colas es el trabajo extra que hay que hacer cuando la cola casi se llena y luego casi se vacía, y finalmente necesitamos guardar más elementos. De una cola de 5 elementos donde se han hecho 4 inserciones y el mismo número de extracciones, los apuntadores *first* y *last* quedan apuntando a la penúltima localidad libre. Físicamente quedan 5 localidad pero ¿podemos utilizarlas todas en este momento? La respuesta es no. Tenemos que copiar los elementos restantes de la cola (y los apuntadores) al principio de ésta para utilizar las localidades libres. Si la cola es muy grande, entonces la operación de copia será muy costosa.

Cola circular. Una cola circular solventa el problema anterior moviendo al apuntador *last* al principio de la cola (siempre y cuando dicha cola no esté llena) dando la apariencia de un movimiento circular. De igual forma, el apuntador *first* se mueve al principio de la cola – de ser necesario – cuando ha llegado al final de la misma. El manejador de la lista circular deberá calcular en cada inserción y extracción el momento en el que alguno de los apuntadores deba "dar la vuelta".

Casos especiales

La implementación particular de la cola debe hacerse cargo de los siguientes casos especiales:

- La cola no existe.
- Se pretende extraer de una cola vacía.
- Se pretende insertar en una cola llena (cuando el concepto de "lleno" tiene sentido).
- Se pretende destruir una cola que no existe.

3.2 Bibliografía

La indicada en el Syllabus de la materia.