Implementaciones de pilas, colas y afines. Memoria dinámica.

Fernando Schapachnik[1]

[1]Departamento de Computación, FCEyN,
Universidad de Buenos Aires, Buenos Aires, Argentina

Algoritmos y Estructuras de Datos II,
segundo cuatrimestre de 2016

(2) Recordemos qué es una pila...

TAD $\textsc{Pila}(\alpha)$

observadores básicos

vacía? : pila(α) $\longrightarrow$ bool

tope : pila(α) p $\longrightarrow$ α ($\neg$vacía?(p))

desapilar : pila(α) p $\longrightarrow$ pila(α) ($\neg$vacía?(p))

generadores

vacía : $\longrightarrow$ pila(α)

apilar : $\alpha \times$ pila(α) $\longrightarrow$ pila(α)

otras operaciones

tamaño : pila(α) $\longrightarrow$ nat

axiomas ($\forall$ *pila*(α) : p, q) , ($\forall$ α : e)

vacía?(vacía) $\equiv$ *true*

vacía?(apilar(e,p)) $\equiv$ *false*

tope(apilar(e,p)) $\equiv$ e

desapilar(apilar(e,p)) $\equiv$ p

tamaño(p) $\equiv$ **if** vacía?(p) **then** 0 **else** 1 + tamaño(desapilar(p)) **fi**

Fin TAD

(3) ...y una cola

TAD $\textsc{Cola}(\alpha)$

observadores básicos

vacía? : cola(α) $\longrightarrow$ bool

próximo : cola(α) c $\longrightarrow$ α ($\neg$vacía?(c))

desencolar : cola(α) c $\longrightarrow$ cola(α) ($\neg$vacía?(c))

generadores

vacía : $\longrightarrow$ cola(α)

encolar : $\alpha \times$ cola(α) $\longrightarrow$ cola(α)

otras operaciones

tamaño : cola(α) $\longrightarrow$ nat

axiomas ($\forall$ *cola*(α) : c, d) , ($\forall$ α : e)

vacía?(vacía) $\equiv$ *true*

vacía?(encolar(e,c)) $\equiv$ *false*

próximo(encolar(e,c)) $\equiv$ **if** vacia?(c) **then** e **else** próximo(c) **fi**

desencolar(encolar(e,c)) $\equiv$ **if** vacía?(c) **then** vacía **else** encolar(e, desencolar(c)) **fi**

tamaño(c) $\equiv$ **if** vacía?(c) **then** 0 **else** 1 +

(4) Notemos

- Notemos que son bastante similares.
- Por ese motivo vamos a trabajar con el TAD COLA.
- A las pilas a veces se las llama *colas LIFO* (last in, first out).
- A las colas, *colas FIFO* (first in, first out).
- En lo que sigue, voy usar un lenguaje que tiene tres características muy importantes:
  - Es una mezcla de Pascal, C y C++ (y tal vez otros).
  - No tiene sintaxis fija ni muy precisa: se adapta mágicamente y varía (también mágicamente) a todas mis necesidades.
  - Para usarlo hay que haber aprobado Algo II.

(5) Una implementación posible

- Haremos nuestra primera implementación utilizando las herramientas que conocemos de Algol: arreglos.
- Diremos que una cola es en realidad un arreglo más un natural para saber su tamaño.
- Convención (para la clase de hoy): las posiciones de un arreglo de n elementos son $0 \ldots n-1$.
- Para este ejemplo, instanciaremos α en `floats`.
- Una posible estructura:

```
struct {
        nat cant;
        float elementos[MAX_CANTIDAD];
} cola;
```

- Idea: Los elementos válidos son los que figuran entre la posición 0 y $cant - 1$.

(6) Una implementación posible (cont.)

- Algunas operaciones son triviales:
  - vacía(c) → `c.cant:= 0;`
  - tamaño(c) → `return c.cant;`
  - vacía?(c) → `return c.cant==0;`
- ¿Cómo encolamos un elemento?
  - encolar(e, c) → `c.elementos[c.cant]:= e; c.cant++;`
- ¿Y el próximo?
  - próximo(c) → `return c.elementos[0];`
- Sólo falta desencolar:

```
i:= 0;
while (i<cant-1)
    c.elementos[i]:= c.elementos[i+1];
    i++;
c.cant--;
```

- ¿Es una buena implementación?
- No, la operación desencolar() es extremadamente "cara". Cuantos más elementos tengamos, más tarda.

(7) Mejorando la implementación

- El problema se podría solucionar si pudiésemos cambiar el próximo, en lugar de suponer que siempre es el elemento 0.
- Nueva propuesta:

```
struct {
        nat cant;
        nat primero;
        float elementos[MAX_CANTIDAD];
} cola;
```

- Muchas operaciones se mantienen, pero otras no.
  - vacía(c) → `c.cant:= 0; c.primero:= 0;`
  - próximo(c) → `return c.elementos[c.primero];`
  - encolar(c) → `c.elementos[c.primero+c.cant]:= e;`
  - desencolar(c) → `c.primero++; c.cant--;`

(8) Mejorando la implementación (cont.)

- Notemos un problema: A diferencia de la implementación anterior, a ésta sólo la voy a poder usar para meter `MAX_CANTIDAD` elementos. En la anterior ése era el límite de elementos que podían convivir en *simultáneo*; acá es el límite total.
- Una posible solución es desplazar los elementos (como hacíamos antes) en algún momento, pero vamos a ver una más interesante.

(9) Aritmética circular

- Introduzcamos el concepto de aritmética circular. ¿Están listos? Miren que es súper novedoso, nunca visto.
- Con ustedes, la artimética circular, alias el *módulo*.
- Idea: las posiciones que están antes de `c.primero` las puedo seguir usando (están vacías).
- La estructura es la misma.
- La creación, tamaño() y vacía?() no varían.
- Muchas operaciones se mantienen, pero otras no.
  - próximo(c) → `return c.elementos[c.primero];`
  - encolar(e, c) →
    ```
    c.elementos[(c.primero+c.cant) % MAX_CANTIDAD]:= e;
    c.cant++;
    ```
  - desencolar(c) →
    ```
    c.primero:= (c.primero+1) % MAX_CANTIDAD;
    c.cant--;
    ```

(10) Problemas...

- Con estas mismas ideas podríamos implementar una secuencia...
- ...pero la operación que elimina un elemento de la misma sería "cara", porque involucraría desplazamientos para no dejar huecos.
- ¿Y si no alcanza el tamaño?
- ¿Y si sobra?
- ¿Y si no lo sabemos a priori?
- Algunos lenguajes proveen arreglos redimensionables, pero no son mágicos:
  - El mecanismo consiste en crear uno más grande y luego copiar los elementos.
  - En seguida veremos cómo hacer la parte de crear uno más grande.

(11) Memoria dinámica

- Hay un concepto que resuelve todos estos problemas y aporta algunas ventajas más.

Se llama memoria dinámica y es de extrema importancia dentro de la programación.

- Además, lo usaremos como base de la mayor parte de las estructuras con las que trabajaremos en la segunda parte de la materia.

También presentaremos otro concepto, que suele estar relacionado aunque es más o menos independiente, que es el de punteros.

(12) Empecemos por las variables...

- Variable matemática: Valor fijo pero desconocido. Ejemplo: $(\forall x)(x + 1 > x)$
- Variable computacional: Objeto que *contiene* un valor.

(13) Variables computacionales

Materialización del concepto de "objeto que contiene un valor":

- "Personalidad ambivalente":
  - Espacio de memoria que contiene un valor: `x:= 3`
  - El valor contenido: `printf("%d%d", x, 3)`

(14) ¿Qué es la memoria?

- Abstractamente, vector de "bytes". Ejemplo: $M[0..64Mb]$
- Las variables tienen un *tipo* que determina, entre otras cosas, su *tamaño*.
- A cada programa en ejecución, le corresponde un fragmento, dado por las variables estáticas que utiliza.

(15) Ejemplo

```
1  int x;
2  float y;
3  int z;
4
5  x= 3;
6  z= 5;
7  y= z+4.3;
8  z= x;
9
10 return y;
```

Si suponemos que un int usa 2 bytes y un float 4, este programa usa 8 bytes de memoria: $M[comienzo..comienzo + 7]$

(16) Hablando de personalidad ambivalente...

- Supongamos que la variable x se almacena en las posiciones $M[0..1]$ y la variable z en $M[6..7]$.
- La línea 8 se interpreta como: poner en las posiciones $M[6..7]$ lo que haya en las posiciones $M[0..1]$.
- Conclusión: podemos utilizar una variable para referirnos a su valor o al espacio de almacenamiento que representa.

(17) ¿A dónde nos lleva esto?

- Hasta ahora vimos variables *estáticas*.
- Existen también las variables *dinámicas*: sirven, entre otras cosas, para los casos en los que no sabemos de antemano el tamaño de la entrada.
- Ejemplo:

```
Preguntarle al usuario la cantidad
      de enteros a ingresar (C).
Reservar memoria para C enteros.
for (i= 1; i<=C; i++)
       Leer y almacenar los enteros.
```

(18) Punteros

- ¿Por qué?
  - Queremos referirnos a posiciones arbitrarias de la memoria.
  - Permite utilizar *estructuras dinámicas*.

(19) Punteros (cont.)

- Sintaxis (à la C):
  - Declaración: `int *p;`
  - Ver el valor al que apunta: `x= *p;`
  - Macro común: `(*p).campo` $\equiv$ `p->campo`
  - Asignarle un valor: `p= 400;` (¡OJO! eso significa que p apunta a $M[400]$)
  - Asignarle un valor: `x= 398; p= x;` (¡OJO! eso significa que p apunta a $M[398]$)
  - Asignarle un valor (¡esta vez bien!): `p= &x;` (p apunta a la "celda" llamada x, es decir, $M[0..1]$)
  - Asignarle un valor a la posición por él apuntada: `*p= 400;`
  - ¿Qué hace `p= &x; *p= &x;`?

(20) Implementación con memoria dinámica y punteros

- Nueva estructura, con punteros.
- Estructura:

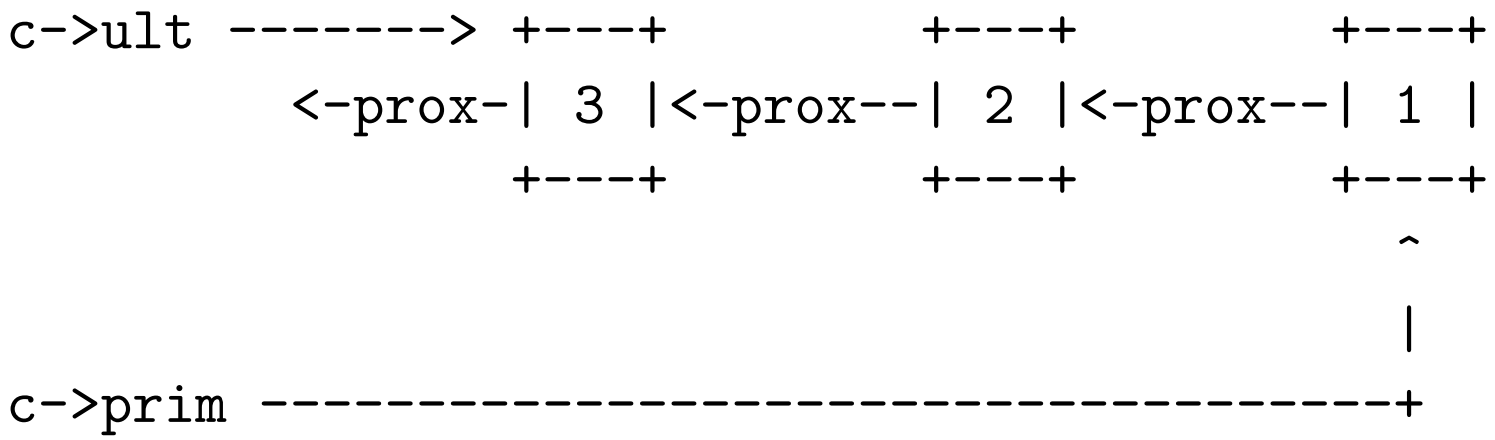
```
struct {
        struct nodo_cola *prim;
        struct nodo_cola *ult;
        nat cant;
} cola;
```

- ¿Qué es un `nodo_cola`?

```
struct nodo_cola {
        float elem;
        struct nodo_cola *prox;
};
```

(21) Implementación con memoria dinámica y punteros (cont.)

- Imaginemos: `encolar(3, encolar(2, encolar(1, vacía())))`.
- Impresión del artista:



- `prim` apunta al primer elemento encolado de los que quedan (el más viejo, el próximo a salir).
- `ult` apunta al último elemento encolado (el más reciente).
- Cada nodo tiene un puntero al anterior.

(22) Operaciones

- Empecemos por las fáciles.
  - vacía(c) → `c.cant:= 0; c.prim:= NULL; c.ult:= NULL`
  - tamaño(c) → `return c.cant;`
  - vacía?(c) → `return c.cant==0;`
  - próximo(c) → `return c.prim->elem;`

(23) Operaciones (cont.)

- Veamos encolar(), pero primero, una auxiliar nuevo_nodo(elemento):

```
struct nodo_cola *nodo;

nodo:= new(struct nodo_cola);
if (nodo==NULL) HACER_ALGO_CON_EL_PROBLEMA();

nodo->prox:= NULL;
nodo->elem:= elemento;

return nodo;
```

(24) Operaciones (cont.)

- Ahora sí, encolar(c, e):

```
struct nodo_cola *nuevo_nodo;
nuevo_nodo:= nuevo_nodo(e);

if (c.prim==NULL)
 // Es el primer nodo.
 c.prim:= nuevo_nodo;
else
 // Antes había otro nodo.
 c.ult->prox:= nuevo_nodo;

// ult siempre apunta al último elemento agregado.
c.ult:= nuevo_nodo;

c.cant++;
```

(25) Operaciones (cont.)

- Veamos desencolar(c):

```
struct nodo_cola *aux;

aux:= c.prim;
// Ahora el "primero" es el que le seguía.
c.prim:= c.prim->prox;
delete aux;

if (c.cant==1)
 c.ult:= NULL;

c.cant-;
```

(26) Casi colofón

- En el caso de la cola es razonable tener un único puntero por nodo porque nos movemos unidireccionalmente.
- ¿Qué pasa con una secuencia donde queremos mayor flexibilidad?
- Para eso existen las listas doblemente enlazadas.
- Veamos qué pinta tienen...
- ...y pensemos en la función que elimina un nodo.

(27) Repaso y perspectiva

- Vimos:
  - Pilas y colas con arreglos.
  - Las limitaciones que eso imponía.
  - Memoria dinámica.
  - Colas en base a listas enlazadas.
- Veremos
  - Próxima teórica: Complejidad.

(28) Tarea

- Imaginemos una cola que se comporte como FIFO o LIFO de acuerdo a un parámetro al constructor.
- ¿Cómo la programarían?