Estructuras de datos abstractas

Pablo R. Ramis

Universidad Nacional de Rosario, Instituto Politécnico, Dto. de Informática,
prramis@ips.edu.ar,
WWW home page: http://informatica.ips.edu.ar

Resumen Habiendo visto *arrays* y *estructuras* podemos acercarnos a programar y modelizar situaciones con mayor complejidad. Para esto diseñaresmos con mayor abstracción los datos siempre teniendo en cuenta dos cosas:

1. Tipos de datos: Especificación de la TDA.
2. Operaciones: implementación de la TDA.

Una *estructura de datos abstracta* nos permite:

- Encapsulamiento
- Modularidad

Estos conceptos se profundizarán mas adelante.

1. Pila

1.1. Definición

Es una colección de elementos, que podrían estar repetidos, donde la posición de cada uno es relevante y la inserción y la eliminación solo se realiza de la posición *tope* o *cima*.

Es común decir que los elementos se agregan o extraen en orden *LIFO*: *Last In First Out*

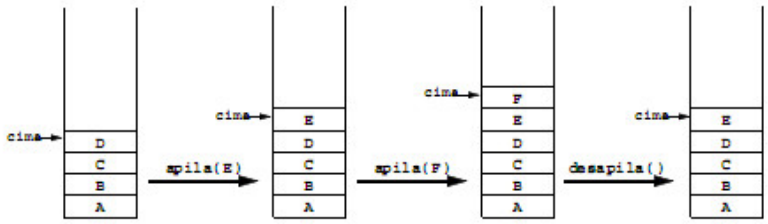

Figura 1. Pila

1.2. Operaciones

Definiremos las operaciones necesarias que debe implementar una pila:

```
1
2  /*agrega un elemento*/
3  apila(e: Elemento);
4
5  /* elimina y retorna el primer elemento */
6  desapila(): Elemento;
7
8  /* retorna el primer elemento pero no lo elimina */
9  cima(): Elemento;
10
11 /* vacia la pila (pasa a tener ñtamao 0) */
12 vaciaPila();
13
14 /* da la cantidad de elementos que posee la pila */ñ
15 tamao(): Entero;
16
17 /* indica si la pila tiene o no elementos */
18 estaVacia(): Booleano;
```

1.3. Implementación de Pilas basadas en arreglos

Necesitamos un *array de elementos* y un *indice* que indica la *cima* o *primer elemento*.

- Cuando la pila esta vacia el indice es -1.
- elementos: Elemento[n]
- cima: Entero

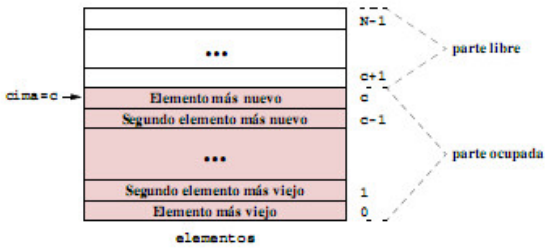

Figura 2. elementos

Veamos en pseudo-código las funciones necesarias:

```
1
2  procedimiento apila(e: Elemento)
3          si ñtamao() = N entonces
4                  error // no caben áms elementos
5          fsi
6          cima := cima + 1
7          elementos[cima] := e
8  fprocedimiento
9
10 procedimiento desapila(): Elemento
11         si áíestVaca() entonces
12                 error  // no hay elementos
13         fsi
14         cima := cima - 1
15         retorna elementos[cima+1]
16 fprocedimiento
17
18 procedimiento vaciaPila()
19         cima := -1
20 fprocedimiento
21
22 procedimiento ñtamao(): Entero
23         retorna cima + 1
24 fprocedimiento
25
26 procedimiento áíestVaca(): Booleano
27         retorna cima = -1
28 fprocedimiento
29
30 procedimiento cima(): Elemento
31         si áíestVaca() entonces
32                 error
33         fsi
34         retorna elementos[cima]
35 fprocedimiento
```

Ejercicio Implementar la pila utilizando *arrays*

```
1  /* pilaArray.h */
2
3  #define CAPACIDAD 15
4
5  int elementos[CAPACIDAD];
6  int cima = -1;
7
```

```
8  void apila(int);
9
10 int desapila();
11
12 void vaciaPila();
13
14 int ñtamao();
15
16 int áíestVaca();
17
18 int cima();
```

```
1  /* pilaArray.c */
2
3  #include<stdio.h>
4  #include "pilaArray.h"
5
6  /* Completar las funciones prototipadas en el .h */
7  int main()
8  {
9
10 /* Armar ócdigo de main para probar la pila */
11
12         return 0;
13 }
```

1.4. Implementación de Pilas basadas en listas simplemente enlazadas

Si bien el concepto es el mismo la implementación podría representarse de esta manera:

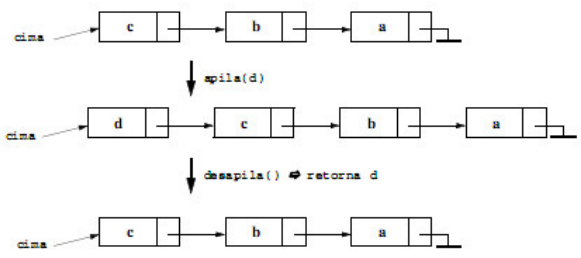

Figura 3. elementos - implementada con lista enlazada

Esto nos da como resultado una implementación mucho más completa no limitándonos en la cantidad de elementos. El concepto de la pila sigue siendo la misma, los elementos se insertarán y extraerán del inicio de la misma.

Una lista simplmente enlazada es aquella estructura en que los nodos tienen un único puntero a otro nodo que es el siguienteo de la lista ademas de los campos de datos propios que posea la estructrua. Además debe contar con una variable que referencie al primer elemento que será el puntero inicio, nuestra via de acceso a la estructura. El campo siguiente del ultimo nodo apunta a NULL

Ejercicio Implementar la pila utilizando *Lista enlazada*

```
/* pilaLista.h */

typedef struct nodo{
        int dato;
        struct nodo* sgte;
}pila;

void apila(pila**, int);

int desapila(pila**);

void vaciaPila(pila**);

int ñtamao(pila*);

int áíestVaca(pila**);

int cima(pila*);
```

```
/* pilaLista.c */

#include<stdio.h>
#include<stdlib.h>
#include "pilaLista.h"

/* Completar las funciones prototipadas en el .h */

int main()
{
        pila *cima;
        cima = NULL;

/* Armar ócdigo de main para probar la pila */

        return 0;
}
```

2. Cola

Colección de elementos potencialmente repetidos donde la posición de cada elemento es relevante, las inserciones se realizan por el *final* y las extracciones por el *inicio*. Es decir el sistema es *FIFO (first in, first out)*.

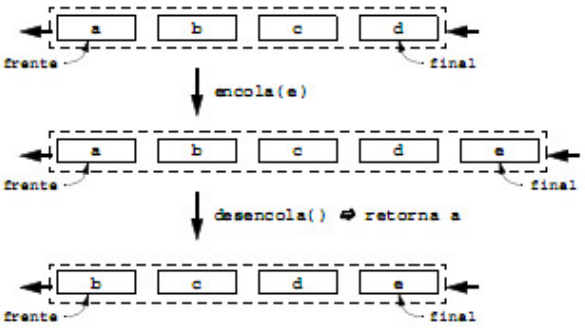

Figura 4. Cola

De forma similar que la *pila*, la *cola* tendrá un conjunto de operaciones básicas que implementan su funcionamiento:

- ***encolar*** que añade un elemento al final de la cola.
- ***desencolar*** elimina y retorna el primer elemento de la cola. El segundo pasa a ser el primero luego de la operación.
- ***frente*** retorna pero no elimina al primer elemento.
- ***vaciaCola*** vacia la cola, el tamañó pasa a ser cero.
- ***tamaño*** retorna el tamaño de la cola, el número de elementos.
- ***estaVacia*** retorna verdadero si esta vacía.

2.1. Implementación de Colas basada en listas enlazadas

Veamos algunos puntos a tener en cuenta para implementarla:

Necesitamos además de la estructura *nodo* similar a la de la pila, dos punteros, *frente* y *final*. Considerar que ambos inicialmente estarán en NULL.

Al momento de encolar, si es el primer elemento que se ingresa, el puntero frente y final referenciarán a este primer nodo. Sino, frente mantendrá su

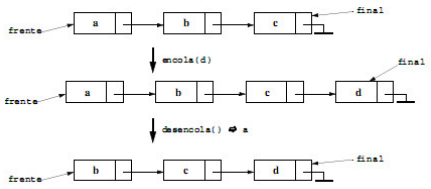

Figura 5. Cola implementada con lista enlazada

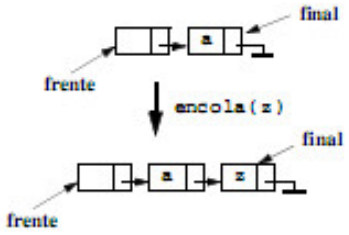

Figura 6. Insertar un elemento a la cola

posición, el elemento nuevo se posicionará como ultimo nodo en la cola y final apuntará a este.

El extraer un elemento es el trabajo con el puntero frente. En primera instacia, se valida que la cola no esté vacía. Luego, se toma el primer elemento, se hace un corrimiento de punteros, o sea frente será igual a frente− >siguiente y al finalizar se retorna el elemento.

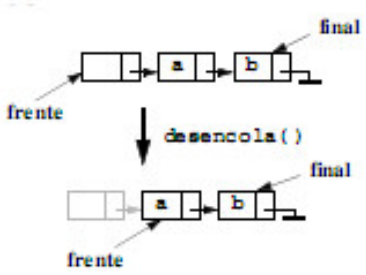

Figura 7. Extrae el primer elemento a la cola

Ejercicio Programar una cola mediante uso de listas enlazadas

```
/* colaLista.h */

typedef struct nodo{
        int dato;
        struct nodo* sgte;
}cola;

void encola(cola**, cola**, int);

int desencola(cola**, cola**);

void vaciaCola(cola**);

int ñtamao(cola*);

int áíestVaca(cola**);

int frente(pila*);
```

```
/* colaLista.c */

#include<stdio.h>
#include<stdlib.h>
#include "colaLista.h"

/* Completar las funciones prototipadas en el .h */

int main()
{
        cola *inicio, *final;
        inicio = NULL;
        final = NULL;

/* Armar ócdigo de main para probar la cola */

        return 0;
}
```

2.2. Cola circular

Comenzaremos aclarando que se ha tomado una forma de implementación de varias posibles, la referencia tomada se encuentra en un excelente libro: ***Estructuras de datos y algoritmos*** de *Alfred Aho, Jeffrey Ullman y John Hopcroft.*

Las colas circulares están implementadas con *arrays* y también tendrán dos índices indicando *frente* y *final.*

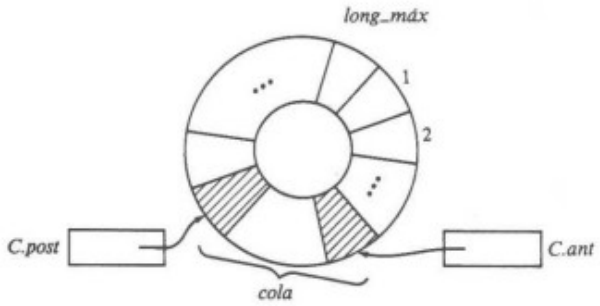

Figura 8. Cola circular

Imaginemos la situación, la cola se encuentra en alguna parte de ese círculo, ocupa posiciones consecutivas, el extremo posterior a la izquierda del anterior. Al insertar un elemento moveremos al apuntador C.pos una posición en sentido a las agujas del reloj y se guarda el elemento en esa posición.

Para suprimir, se mueve el cursor C.ant en sentido del reloj. De esta manera la cola se mueve siempre en el mismo sentido y facilita mucho los movimientos.

Si vemos igualmente esta forma de implementación, hay un detalle a tener en cuenta: C.pos apunta una posición más adelantada al último elemento. El problema en esto radica que no hay forma de distinguir cuando la cola esta llena o vacia a menos que se use un bits para indicar esta condición, sino, la única forma de evitar esto es que nunca se llene la cola.

Si analizamos lo que ocurre, imaginemos una cola con *longMax* elementos, C.pos debería estar en una posición adelante de C.ant en sentido contrario a las agujas del reloj. ¿Qué ocurre si la pila esta vacía? Imaginemos que tiene solo un elemento, C.ant y C.pos apuntan al mismo lugar, si en esta condición se suprime un elemento, C.ant avanza una posición en sentido de las agujas del reloj formando la cola vacía, y C.pos quedó a una posición de C.ant en el sentido contrario a las agujas del reloj o sea, la misma posición que tendría si la cola contuviera *longMax* elementos, esto muestra que aunque la capacidad sea *longMax* solo se podrán admitir $longMax - 1$ a menos que se utilice como se mencionó al principio un mecanismo para identificar la situación.

Implementación Se mostrará la implementación en pseudocódigo:

```
1
2   procedure suma_uno ( i: integer ): integer;
3           begin
4               return ((i mod long_max) + 1)
5           end; { suma_uno }
6
```

```
7       procedure ANULA ( var C: COLA);
8            begin
9                C.ant := 1;
10               C.pos := long_max
11           end; { ANULA }
12
13      function VACIA ( var C: COLA): boolean;
14           begin
15               if suma_uno(C.pos) = C.ant then
16                    return (true)
17               else
18                    return (false)
19           end; { VACIA }
20
21      function FRENTE ( var C: COLA ): tipo_elemento;
22           begin
23               if VACIA(C) then
24                   error('cola ívaca')
25               else
26                   return (C.elementos[C.ant])
27           end; { FRENTE }
28
29      procedure ENCOLAR ( x: tipo_elemento; var C: COLA );
30           begin
31               if suma_uno(suma_uno(C.pos)) = C.ant then
32                    error('La cola esta llena')
33               else begin
34                    C.pos := suma_uno(C.pos);
35                    C.element[C.pos] := x
36            end; { ENCOLAR }
37
38      procedure DESCOLAR ( var C: COLA );
39           begin
40               if VACIA(C) then
41                   error('cola vacia')
42               else
43                   C.ant := suma_uno(C.ant)
44           end; { DESCOLAR }
```

Ejercicio Implementar una cola circular

3. Listas enlazadas

Si bien podemos considerar que tanto la pila como la cola son casos especiales de una lista enlazada, la hemos dejado para el final de estas estructuras para considerarlas con unas condiciones particulares, nosotros las implementaremos con las siguientes caracteristicas:

- La lista siempre estará ordenada, sea de forma ascendente o descendente
- Puedo extraer a cualquier elemento de la lista sin importar la ubicación que posea.

Para ilustrar:

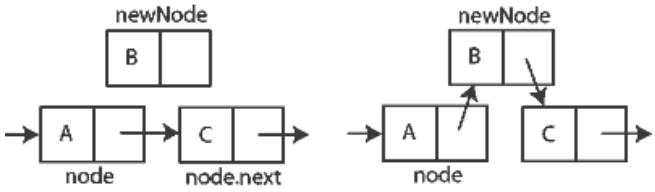

Figura 9. Insertar elemento en la lista

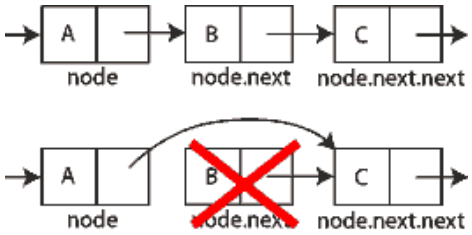

Figura 10. borrar elemento de la lista

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

typedef struct nodo{

    int dato;
    struct nodo * sgte;
}lista;

void insertar (int d, lista** i){

    if (*i == NULL){
        printf("Insertando %d en la lista\n", d);
        *i = (lista*) malloc(sizeof(lista));
        (*i)->dato = d;
    }else{
        if ((*i)->dato > d){ // cuando va al inicio
```

```
18              printf("Insertando %d en la lista\n", d);
19              lista* nuevo = (lista*) malloc(sizeof(lista));
20              nuevo->dato = d;
21              nuevo->sgte = *i;
22              *i = nuevo;
23          }else if ((*i)->dato < d && (*i)->sgte != NULL && (*i
                )->sgte->dato > d){ //cuando esta entre dos
24              printf("Insertando %d en la lista\n", d);
25              lista* nuevo = (lista*) malloc(sizeof(lista));
26              nuevo->dato = d;
27              nuevo->sgte = (*i)->sgte;
28              (*i)->sgte = nuevo;
29          }else if ((*i)->dato == d)
30              printf("Ya existe el elemento %d en la lista\n",
                    d);
31          else
32              insertar(d, &(*i)->sgte);
33      }
34
35  }
36
37  void listar(lista* i){
38
39      if (i != NULL){
40          printf("elemento := %d \n", i->dato);
41          listar(i->sgte);
42      }
43
44  }
45
46  void eliminar(int d, lista** i){
47
48      if (*i == NULL)
49          printf("No existe el elemento a eliminar\n");
50      else{
51          if ((*i)->dato == d){ //elimino al primero
52              printf("Elimino a %d\n", d);
53              lista* aux = (*i);
54              (*i) = (*i)->sgte;
55              free(aux);
56          }else if((*i)->dato < d && (*i)->sgte != NULL && (*i)
                ->sgte->dato == d){ //elimino al siguiente
57              printf("Elimino a %d\n", d);
58              lista* aux = (*i)->sgte;
59              (*i)->sgte = (*i)->sgte->sgte;
60              free(aux);
61          }
62          else
63              eliminar(d, &(*i)->sgte);
64      }
```

```
65  }
66
67  int main (){
68      lista * inicio;
69      inicio = NULL;
70
71      insertar(5, &inicio);
72      insertar(3, &inicio);
73      insertar(4, &inicio);
74      insertar(10, &inicio);
75      insertar(6, &inicio);
76      insertar(8, &inicio);
77      insertar(1, &inicio);
78      insertar(8, &inicio);
79      insertar(1, &inicio);
80
81      printf("Listo elementos\n");
82      listar(inicio);
83      printf("Fin del listado\n");
84
85      eliminar(4, &inicio);
86      eliminar(1, &inicio);
87      eliminar(10, &inicio);
88      eliminar(14, &inicio);
89
90      printf("Listo elementos\n");
91      listar(inicio);
92      printf("Fin del listado\n");
93
94      return 0;
95  }
```

Ejercicio Teniendo en cuenta lo visto antes, implementar una lista doblemente enlazada.

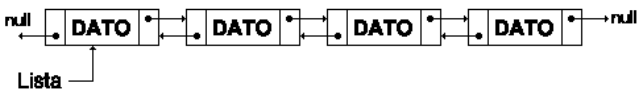

Figura 11. Lista doblemente enlazada

4. Arboles binarios

Dentro de las estructuras de diccionarios, el árbol es una de las más comunes, hay una cantidad importante de variantes posibles en su implementación, no es ahora lo que analizaremos. Nos enfocaremos especialmente en un tipo particular de árbol, este se compone de nodos, uno se distingue por ser *raiz*, o sea, el primero, y podrá tener dos nodos dependiendo de el: *hijo_izquierdo* e *hijo_derecho*, de menor y mayor valor que él respectivamente.

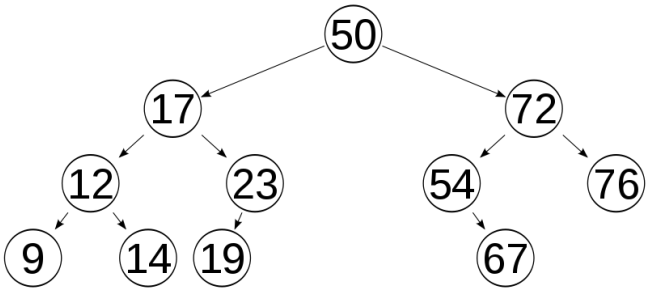

Figura 12. Arbol Binario

Como todas estas responden a unas operaciones básicas:

- Insertar
- Eliminar
- Es Miembro

La búsqueda se simplifica debido a la estructura ordenada del árbol. Si el elemento buscado no es el miembro raíz, verifico si es mayor o menor que este, en el primer caso voy a la derecha sino a la izquierda, y se repite el procedimiento con la nueva ubicación.

En el caso de eliminar, es mas complejo, ya que las opciones son mayores y en ningún caso se debe dejar al árbol inconexo.

Veremos el código para implementarlo

```
1  /* arbolbinario.h */
2
3  #define long_max 25;
4
5  typedef struct nodo{
6          int dato;
7          struct nodo *h_izq, *h_der;
8  }arbol;
9
10 void inserta(arbol** , int);
```

```
11
12  int es_miembro (arbol*, int);
13
14  int suprime_minimo (arbol**);
15
16  void suprime (arbol**, int);
17
18  void listar (arbol*);
```

```
1   /* arbolbinario.c */
2
3   #include<stdio.h>
4   #include<stdlib.h>
5   #include "arbolbinario.h"
6
7
8   void inserta(arbol **A, int x)
9   {
10      if (*A == NULL){
11          *A = (arbol*)malloc(sizeof(arbol));
12          (*A)->dato = x;
13          (*A)->h_izq = NULL;
14          (*A)->h_der = NULL;
15      }
16      else{
17          if (x < (*A)->dato)
18              inserta (&((*A)->h_izq), x);
19          else if(x > (*A)->dato)
20              inserta (&((*A)->h_der), x);
21      }
22  }
23
24  int es_miembro(arbol* A, int x){
25      if (A == NULL)
26          return -1;
27      else if (A->dato == x)
28          return 1;
29      else if (A->dato > x)
30          return es_miembro (A->h_izq, x);
31      else
32          return es_miembro (A->h_der, x);
33  }
34
35  int main()
36  {
37      arbol *raiz;
38      raiz = NULL;
39
```

```
40  /* Armar ódigo de main para probar el arbol */
41
42      inserta(&raiz, 10);
43      inserta(&raiz, 5);
44      inserta(&raiz, 18);
45      inserta(&raiz, 16);
46      inserta(&raiz, 3);
47
48      if (es_miembro(raiz, 5) == 1)
49          printf("el elemento buscado existe en el arbol\n");
50      else
51          printf("No existe \n");
52
53      if (es_miembro(raiz, 124) == 1)
54          printf("el elemento buscado existe en el arbol\n");
55      else
56          printf("No existe \n");
57
58      return 0;
59  }
```

Nos está quedando el ver las funciones para eliminar, por los prototipos que están definidos vemos que son dos. Analizaremos las posibles situaciones a tener en cuenta para borrar un elemento teniendo en cuenta los siguientes puntos:

1. Hay que encontrar el elemento a borrar.
2. Si es uno de los ultimos nodos, o sea, un hijo o como se suele llamar una *hoja* sin nodos dependientes de él, no hay complicaciones, podremos borrarlo en forma directa.

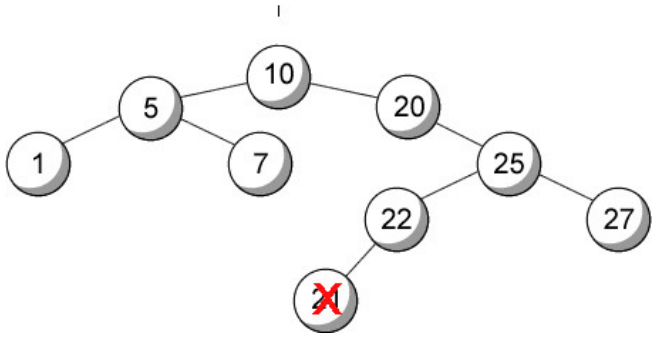

Figura 13. Eliminar una hoja

3. Si decidieramos borrar un nodo con hijos, se debe tener la precaución de no dejar inconexo al arbol y que este continue con el orden correspondiente. Para

lograr esto, se reemplazará por el nodo con el valor mas pequeño de la rama derecha del nodo a eliminar. En este caso en particular, se reemplazará el 20

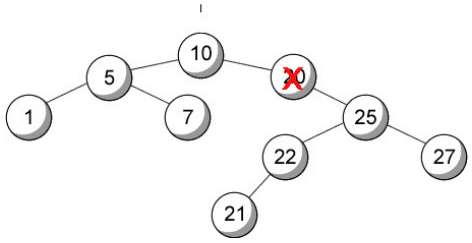

Figura 14. Eliminar nodo intermedio

por el 21.

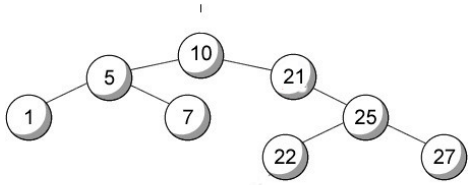

Figura 15. Estado final después de la eliminación

El código será de la siguiente manera:

```
1
2  int suprime_min(arbol **A)
3  {
4      int v_ref;
5      if ((*A)->h_izq == NULL){
6          v_ref = (*A)->dato;
7          arbol *tmp = *A;
8          *A = (*A)->h_der;
9          free(tmp);
10         return v_ref;
11     }else{
12         return suprime_min(&((*A)->h_izq));
13     }
14
```

```
15  }
16
17  void suprime(arbol** A, int x){
18
19      if (*A != NULL){
20          if (x < (*A)->dato)
21              suprime(&((*A)->h_izq), x);
22          else if (x > (*A)->dato)
23              suprime(&((*A)->h_der), x);
24      // Lo encontre
25          else if((*A)->h_izq == NULL && (*A)->h_der == NULL){
26              arbol *tmp = *A;
27              *A = NULL;
28              free(tmp);
29          }else if ((*A)->h_izq == NULL){
30              arbol * tmp = *A;
31              *A = (*A)->h_der;
32              free(tmp);
33          }else if ((*A)->h_der == NULL){
34              arbol *tmp = *A;
35              *A = (*A)->h_izq;
36              free(tmp);
37          }else{ //ambos hijos áestn presentes
38              (*A)->dato = suprime_min(&((*A)->h_der));
39          }
40      }
41  }
```

Ejercicio:

1. Implementar un árbol binario que soporte strings de caracteres.
2. Implementar tres funciones para el listado del árbol: *pre_orden*, *en_orden* y *pos_orden*