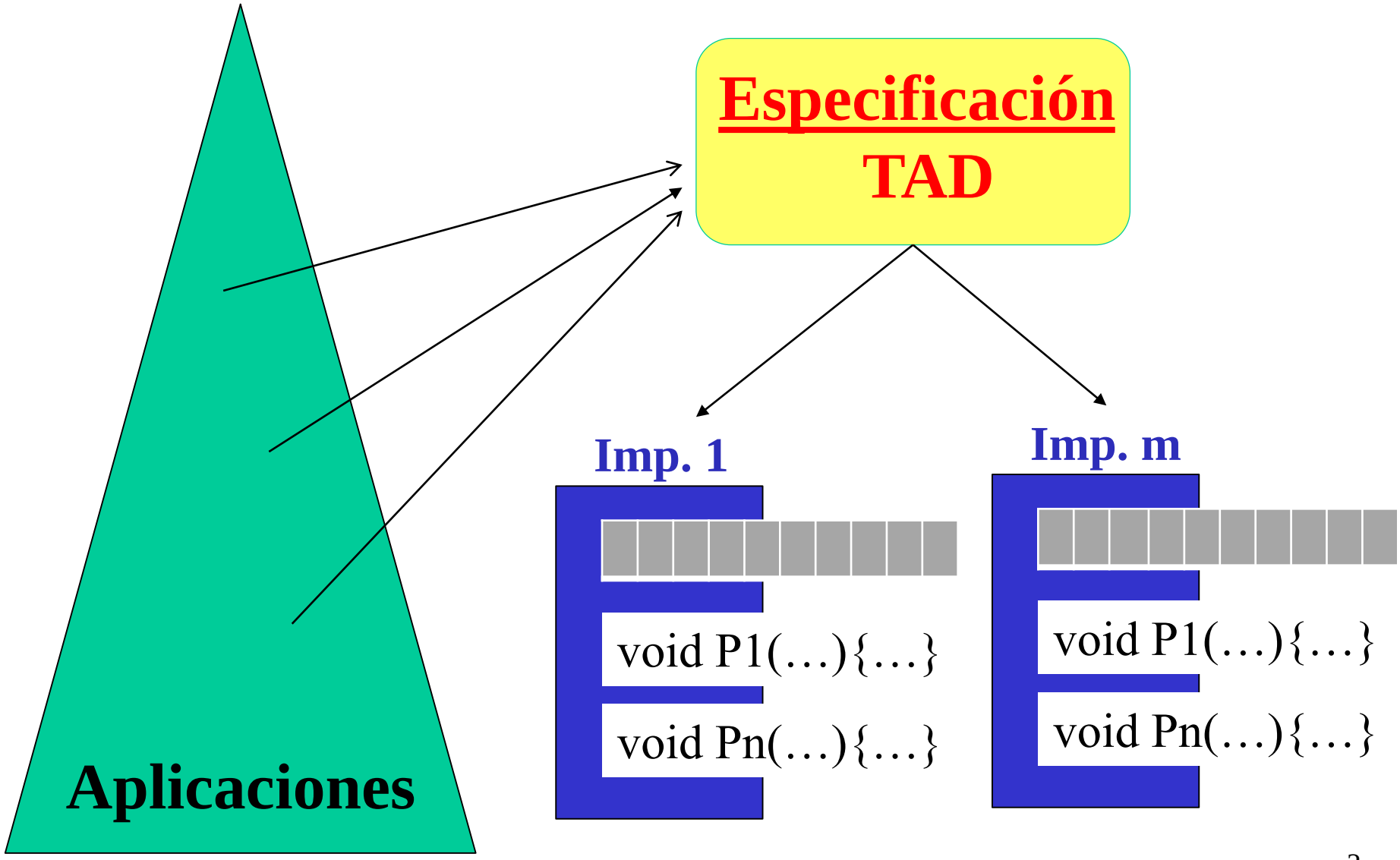
Programación 2

Introducción a TADs
Lista – Pila – Cola

Sobre TADs


Especificación
TAD
Aplicaciones
Imp. 1
void P1(…){…}
void Pn(…){…}
Imp. m
void P1(…){…}
void Pn(…){…}

TAD LISTA

Definición

Hemos visto la definición inductiva de listas.

Más precisamente, definimos inductivamente los tipos Alista, para cualquier tipo A (que es el tipo de los elementos de las listas en cuestión).

Los tipos Alista se definen inductivamente por medio de dos constructores:

[ ] : Alista

__ . __ : A * Alista -> Alista

Conceptos básicos

En esta notación, indicamos los tipos de las operaciones: qué argumentos requieren y qué tipo de objetos producen.

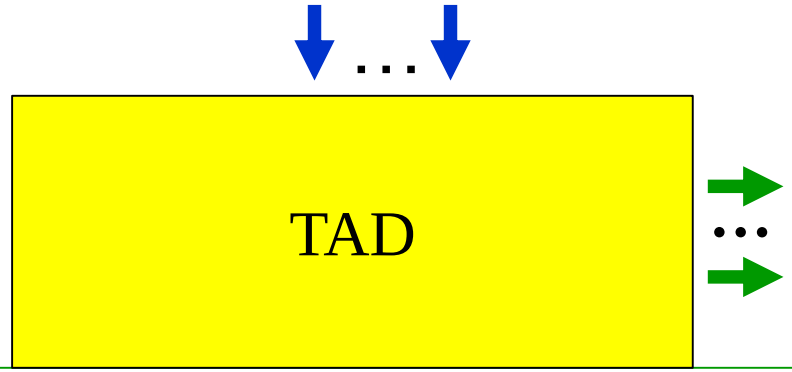
Se entiende entonces que, por definición, las Alistas son todos aquellos objetos que pueden ser formados combinando estas dos operaciones.

Ahora tratamos de presentar las listas como un TAD.

Especificación suficiente

El método de selección de las operaciones constructoras, predicados, selectoras/destructoras puede aplicarse a todo tipo definido inductivamente.

El resultado es una especificación suficiente, en el sentido que toda otra operación sobre listas puede definirse en términos de las primitivas.



Observación

No existe un único TAD Lista (una única especificaión de Lista).

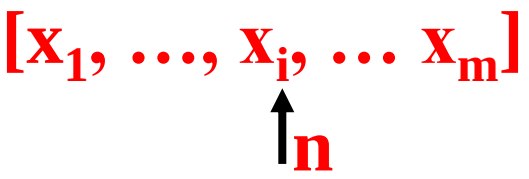
Para Listas, como para muchos TADs, existe más de una especificación, con operaciones diferentes.

Por ejemplo, si consideramos la inserción de elementos en una lista podríamos incluir operaciones que insertan:

- al comienzo de una lista o al final (lugares fijos)
- en una posición dada (listas indizadas o de posiciones explícitas),
- luego de una posición corriente (lista de posiciones implícitas),
- de manera ordenada (listas ordenadas).

Ejemplo de una especificación de Lista

Considere el TAD **Lista Indizada no acotada** de elementos de un tipo genérico, con las siguientes operaciones:



Constructoras:

Lista_Vacia: construye la lista vacía.

Insertar: dados una lista, un entero n y un elemento e, inserta e en la lista en la posición n. Si la lista tiene longitud m, con m menor a n-1, lo inserta en la posición m+1. Si la lista tiene longitud m, con m mayor o igual a n-1, inserta e en la posición n y desplaza en una posición los elementos que estuvieran en las posiciones siguientes.

Predicados:

Esta_Vacia: retorna true si, y solamente si, la lista es Vacía.

Esta_Definido: dados una lista y un entero n, retorna true si, y solamente si, la lista está definida en la posición n.

Ejemplo de una especificación de Lista (cont.)

$[x_1, \ldots, x_i, \ldots x_m]$

n

Constructoras:

Selectoras:

Elemento: dados una lista y un entero n, retorna el elemento en la posición n. Si la lista tiene longitud menor a n, la operación está indefinida.

Borrar: dados una lista y un entero n, elimina de la lista el elemento en la posición n. Si la posición no está definida, la operación no hace nada. Si la posición está definida, elimina el elemento en dicha posición y desplaza en una posición los elementos que estuvieran en las posiciones siguientes (contrae la lista).

Destructora:

Destruir: destruye una lista, liberando la memoria que ésta ocupa.

¿Cuáles serían las precondiciones y cuáles las postcondiciones?

Más operaciones en una especificación

Puede ser útil en los TADs contar con operaciones adicionales. En particular para:

- crear una copia (clon) del TAD sin compartir memoria con éste.
- Saber la cantidad de elementos que tiene el TAD.

Implementación

Una implementación de un TAD consiste en un tipo de estructura de datos concreto que se elige como representación del TAD y las correspondientes implementaciones de las operaciones, que respetan sus postcondiciones.

En el caso de las listas especificadas previamente, una implementación natural se basa en la representación por medio de listas encadenadas.

Manejo de Precondiciones

Alternativas para manejar precondiciones en las operaciones:

Al implementar una operación con precondición, asumir que la misma se cumple (la responsabilidad es del usuario del TAD).

Considerar casos erroneos (por ejemplo, con un “if”). Esto lleva a totalizar una operación parcial. El implementador realiza el manejo de los errores en el uso de las precondiciones.

Manejo de Precondiciones

Uso de la macro assert de C/C++.

assert (*expresion*);

prueba el valor de una expresión. Si el valor de la expresión es 0 (falso), entonces assert imprime un mensaje de error (conteniendo el número de línea y el nombre del archivo), y llama a la función *abort* para terminar la ejecución del programa.

Esta es una herramienta de depuración útil.

TAD´s acotados y no acotados

Los TAD´s pueden ser acotados o no en la cantidad de elementos. **En la especificación de los TAD´s acotados**:

- La operación que crea el TAD vacío recibe como parámetro la cantidad máxima de elementos admitida.
- Se incorpora una operación (un predicado) que permite testar si el TAD está lleno.
- Se agrega también una precondición a la especificación de las operaciones de inserción del TAD, ya que éste no puede estar lleno antes de insertar.

Estas son las diferencias que existen entre la versión acotada y no acotada de un TAD. **Notar que hablamos siempre de especificación, ya que la implementación no define comportamiento, solo lo implementa!**

TAD´s acotados y no acotados

En general, las implementaciones **estáticas** refieren a una versión del TAD que es acotada, mientras que las implementaciones **dinámicas** admiten una versión del TAD no acotada (que permite expresar la noción de estructuras arbitrariamente grandes).

¿Tiene sentido una implementación usando estructuras dinámicas para una Lista acotada?

¿Tiene sentido el uso de un vector (por ejemplo) para implementar una Lista no acotada?

Implementaciones de Listas

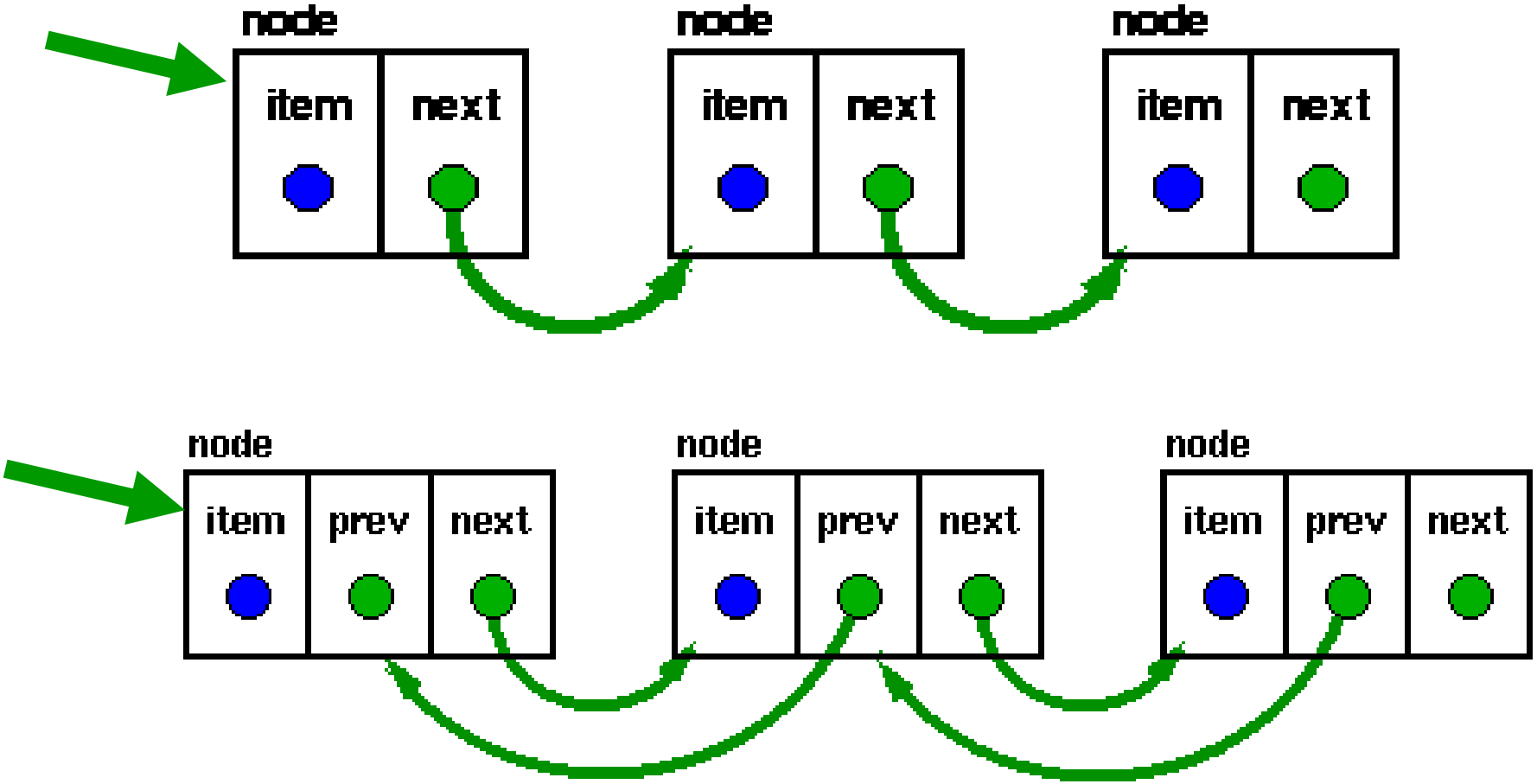
NOTA:

En el práctico se trabajará sobre especificaciones y en particular implementaciones de listas:

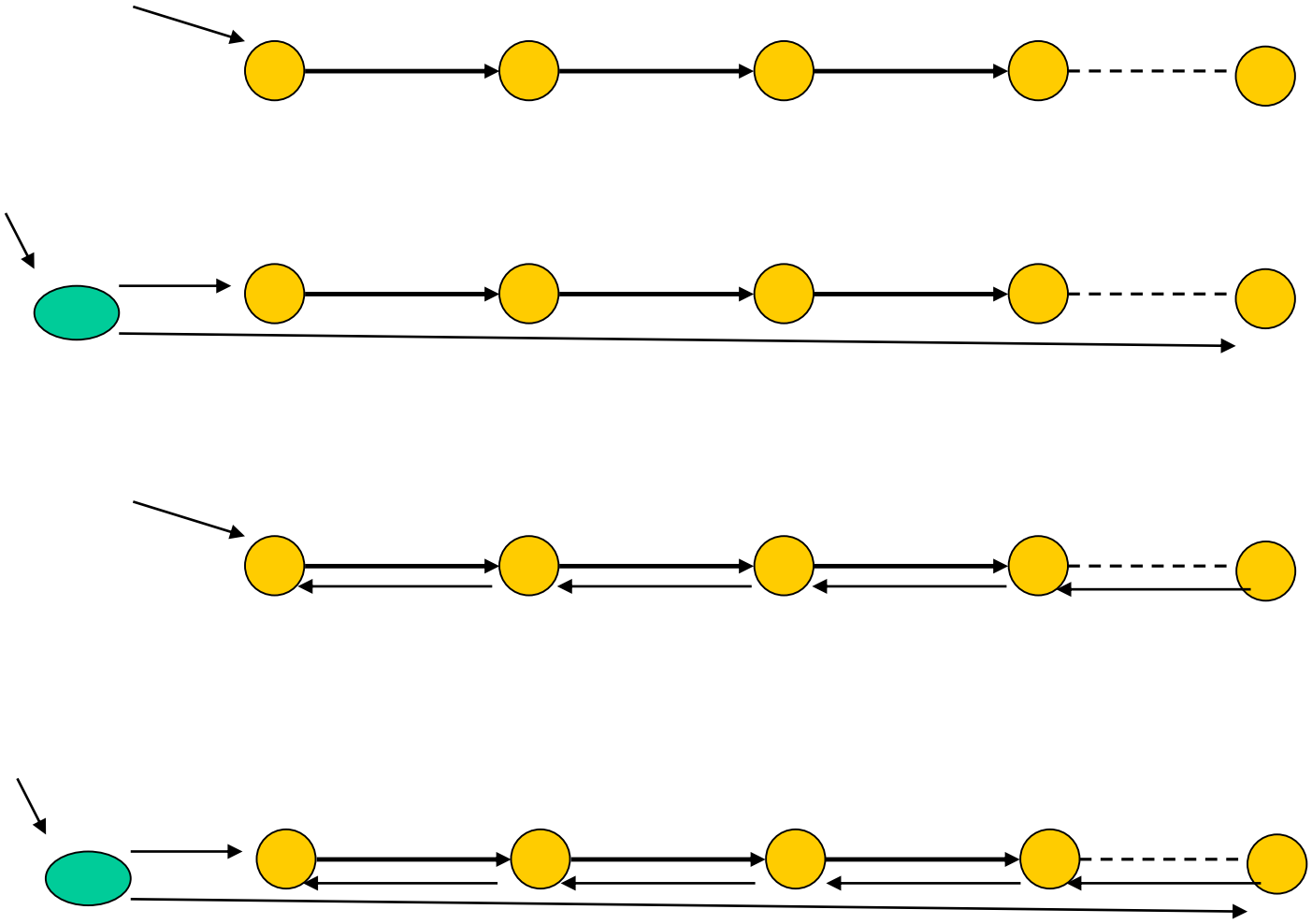
– dinámicas, como por ejemplo listas simple y doblemente encadenadas, circulares, y

– (pseudo)estáticas, por ejemplo con el uso de vector (arreglos).

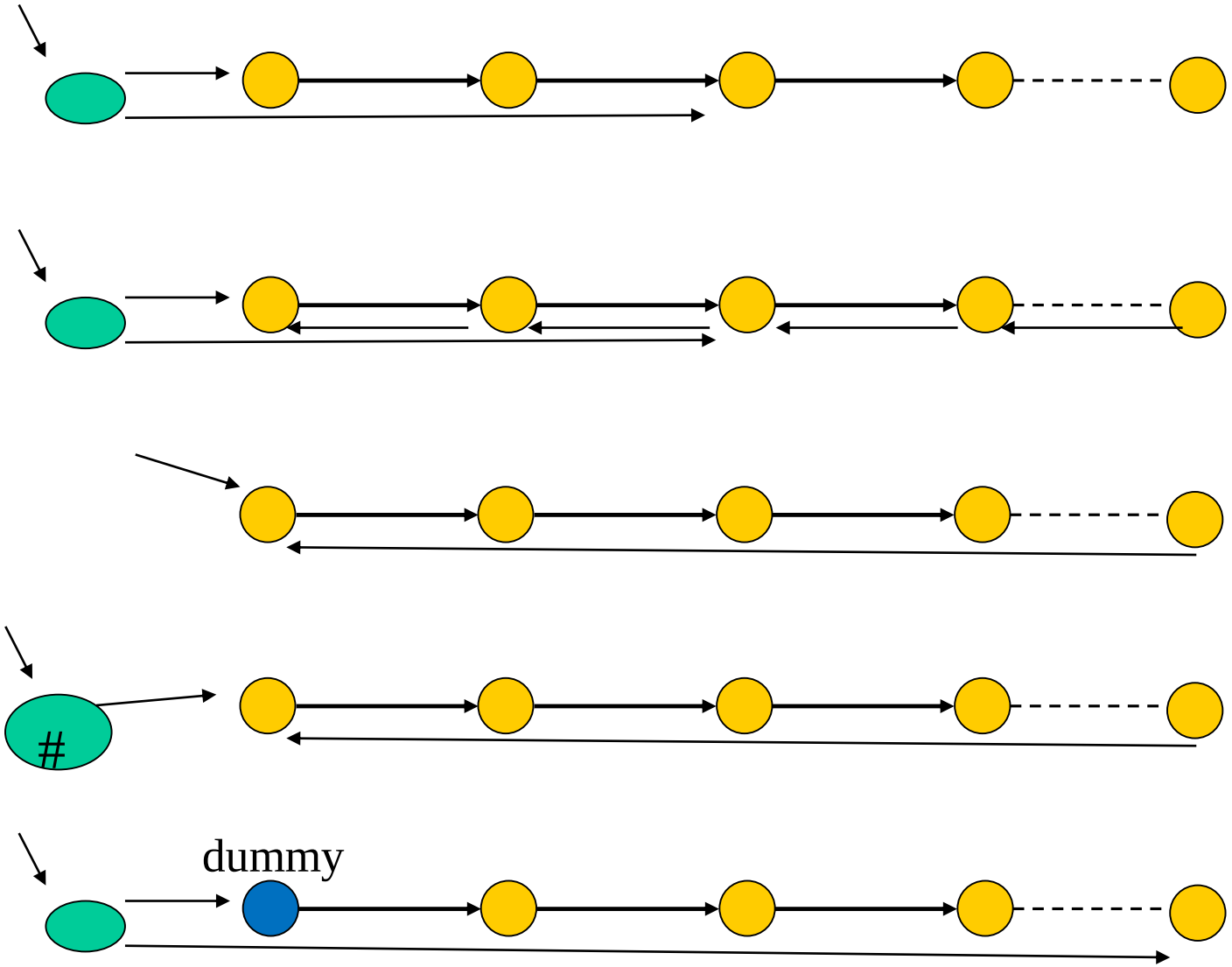
Implementaciones dinámicas de Listas



Algunas variantes de Listas



Algunas variantes de Listas



Ejercicios sobre el TAD Lista

Especifique en C el TAD *Lista Indizada no Acotada* descrito previamente.

Implemente el TAD usando listas de memoria dinámica simplemente encadenadas.

Ejercicios sobre el TAD Lista
Lista no acotada de posiciones implícitas

Especifique en C++ el TAD *Lista no acotada de posiciones implícitas* de elementos de un tipo T, con operaciones para*:*

- Crear la lista vacía;
- Insertar un elemento luego de la posición actual. Si la lista es vacía, se agrega el elemento. Si la posición actual es la del último elemento, se agrega el nuevo elemento al final de la lista. La posición actual pasa a ser en cualquier caso la del elemento insertado;
- Insertar un elemento antes de la posición actual. Si la lista es vacía, se agrega el elemento. Si la posición actual es la del primer elemento, se agrega el nuevo elemento al comienzo de la lista. La posición actual pasa a ser en cualquier caso la del elemento insertado;

Ejercicios sobre el TAD Lista

Lista no acotada de posiciones implícitas

- Fijar la posición actual al comienzo de la lista (en el primer elemento), si ésta no es vacía;
- Avanzar la posición actual al próximo elemento de la lista, si ésta no es vacía y la posición actual no corresponde a la del último elemento;
- Chequear si la lista es vacía;
- Chequear si la posición actual es la del último elemento (se encuentra al final de la lista), si la lista no es vacía;
- Retornar el elemento en la posición actual, si la lista no es vacía;
- Eliminar el elemento en la posición actual, si la lista es no vacía. La posición actual pasa a ser la del elemento siguiente en la lista. Si se elimina el último elemento, la posición actual pasa a ser la del ultimo elemento de la lista resultante, si ésta es no vacía.

Ejercicios sobre el TAD Lista
Lista no acotada de posiciones implícitas

Implemente en C el TAD *Lista no acotada de posiciones implícitas* de elementos de un tipo T de tal manera que todas las operaciones tengan O(1) de tiempo de ejecución en el peor caso. Se sugiere utilizar una estructura de lista dinámica doblemente encadenada.

TAD PILA (STACK)

Definición

Un **stack (o pila)** es una clase especial de lista en la que todas las inserciones y supresiones de elementos se efectúan sobre uno de sus extremos, llamado el ***tope*** del stack.

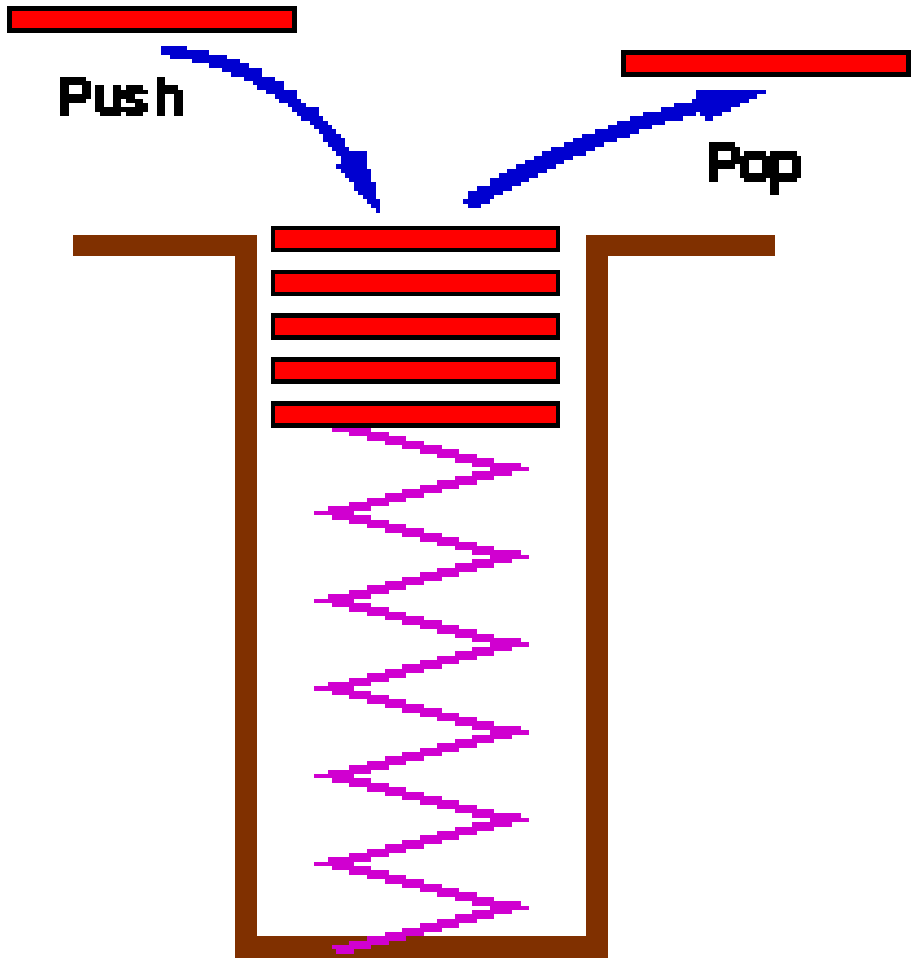
Ejemplos de stacks son:

- pila de fichas de poker en una mesa
- pila de platos para lavar, pila de libros, ...
- stack de ejecuciones

donde es claramente conveniente quitar el elemento que está en el tope de la pila o agregar un elemento nuevo sobre el tope de la misma.

Definición

Otro nombre que se le da a este tipo de estructura es el de lista ***LIFO*** (last-in-first-out).



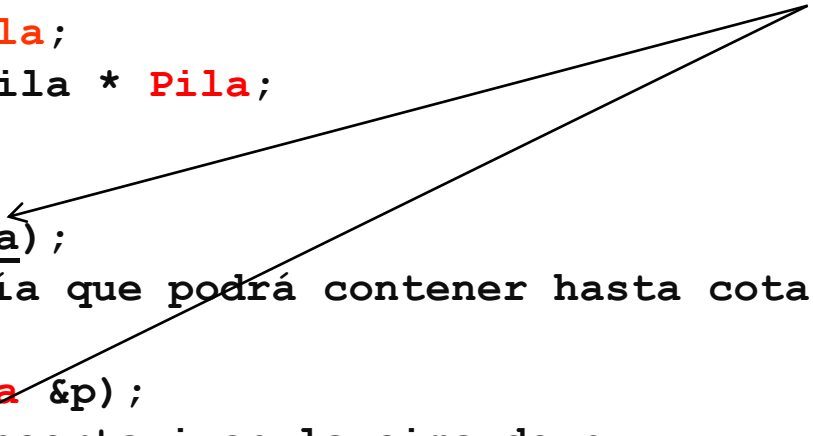
Operaciones

El TAD Stack (Pila) incluye básicamente las siguientes operaciones:

- la operación que construye un stack vacío, **`Empty`** (**`crearPila`**);
- **`Push`** (**`apilar`**), que inserta un elemento en el tope del stack;
- **`Top`** (**`cima`**), que retorna el elemento que se encuentra en el tope del stack;
- **`Pop`** (**`desapilar`**), remueve el elemento que se encuentra al tope (en la cima) del stack;
- **`IsEmpty`** (**`esVaciaPila`**), que testea si el stack es vacío o no;
- Si se trata de un stack acotado se incluye un predicado adicional, **`IsFull`** (**`esLlenaPila`**), que testea si el stack está lleno. Notar que en este caso la operación **`Push`** tendría precondición;
- Finalmente puede incluirse una operación para destruir un stack (**`destruirPila`**), liberando la memoria que éste ocupa.

Especificación del TAD Pila acotada en C/C++

```
#ifndef _PILA_H
#define _PILA_H

struct RepresentacionPila;
typedef RepresentacionPila * Pila;

// CONSTRUCTORAS
Pila crearPila (int cota);
// Devuelve la pila vacía que podrá contener hasta cota elementos.

void apilar (int i, Pila &p);
/* Si !esLlenaPila(p) inserta i en la cima de p,
   en otro caso no hace nada. */

// SELECTORAS
int cima (Pila p);
/* Devuelve la cima de p.
   Precondicion: ! esVaciaPila(p). */

void desapilar (Pila &p);
/* Remueve la cima de p.
   Precondicion: ! esVaciaPila(p). */
```

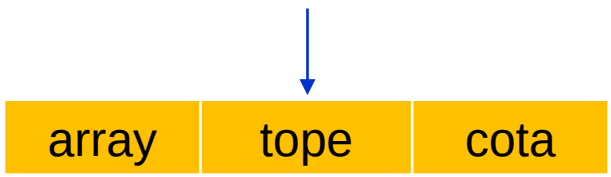


Especificación del TAD Pila acotada en C/C++

```
// PREDICADOS
bool esVaciaPila (Pila p);
/* Devuelve true si y sólo si p es vacia. */

bool esLlenaPila (Pila p);
/* Devuelve 'true' si y sólo si p tiene cota elementos, donde cota
   es el valor del parámetro pasado en crearPila. */

// DESTRUCTORA
void destruirPila (Pila &p);
/* Libera toda la memoria ocupada por p. */


#endif /* _PILA_H */
```

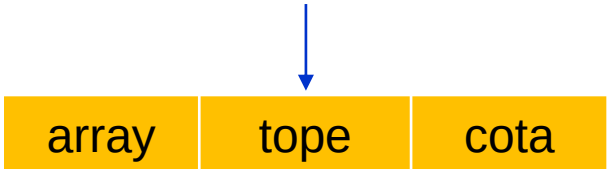
Implementación con un vector de una Pila acotada de enteros en C/C++

```
#include <stddef.h>
#include <assert.h>

#include "pila.h"

struct RepresentacionPila{
   int* array;
   int tope;
   int cota;
};

// CONSTRUCTORAS
Pila crearPila (int cota) {
   Pila p = new RepresentacionPila();
   p->tope = 0;
   p->array = new int [cota];
   p->cota = cota;
   return p;
}
```



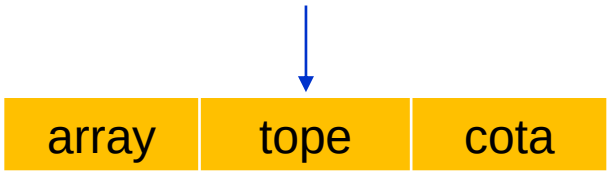
Implementación (cont.)

```
void apilar (int i, Pila &p) {
   if (! esLlenaPila (p)) {
        p->array [p->tope] = i;
        p->tope ++;
   }
}

// SELECTORAS
int cima (Pila p) {
   assert (p->tope > 0);
   return p->array [p->tope - 1];
}

void desapilar (Pila &p) {
   assert (p->tope > 0);
   p->tope--;
}
```



Implementación (cont.)

```
// PREDICADOS
bool esVaciaPila (Pila p) {
   return (p->tope == 0);
}

bool esLlenaPila (Pila p) {
   return (p->tope == p->cota);
}

// DESTRUCTORA
void destruirPila (Pila &p) {
   delete [] p->array;
   delete p;
}
```



Ejemplo de uso del TAD Pila

```
#include <stdio.h>
#include "pila.h"
void main{
   Pila p;
   p = crearPila(10);
   for (int i = 1, i<= 5, i++)
       apilar(i, p);
   while (!esVaciaPila(p)){
            printf("%d\n", cima(p))
            desapilar(p);
   }
   destruirPila(p);
}
```

Notar que este código no depende de la implementación de la pila

Algunas Aplicaciones

Dos aplicaciones interesantes del TAD Pila son:

- En el parsing de expresiones algebraicas y construcciones sintácticas (balanceo…).
- En la implementación de modelos de ejecución de llamados a procedimientos (stack de ejecuciones).

Balanceo de paréntesis

Un compilador para un lenguaje de programación, entre otras cosas, verifica la corrección sintáctica de programas escritos en ese lenguaje.

Frecuentemente la ausencia de una “{”, por ejemplo, puede hacer generar a un compilador una larga lista de diagnósticos, no identificando, sin embargo, el error real.

Una herramienta muy útil en esta situación sería un procedimiento que verifique si el programa a compilar es sintácticamente balanceado.

Aplicación (cont.)

Obviamente no es muy conveniente escribir un programa sólo para resolver este problema, pero veremos que es muy fácil, haciendo uso de un stack, resolver problemas de este tipo.

Problema:

Dada una lista de caracteres que sólo puede contener los elementos **(**, **)**, **[**, **]**, **{** y **}**, deseamos construir un procedimiento que verifique que la expresión es balanceada, donde, por ejemplo, las secuencias …**[…(…)…]…** y …**(…)…{…}…** son correctas, pero …**[…(…]…)** y …**{…(…** no lo son.

Aplicación (cont.)

Ejemplos:

{, {, [,], (,), }, }	Bien
{, },], …	Mal
{, {, [, }, …	Mal
{, {, [,]	Mal

Idea: usar un stack para ir almacenando los símbolos de apertura.

Ver en los 4 ejemplos el uso del stack:

Stack = {

Pseudocódigo del algoritmo que chequea balanceo de símbolos

Construya un stack vacío.

Si la lista no es vacía, obtenga su primer elemento y:

Si es un símbolo de apertura: (, [, {, agréguelo al stack.

Si es un símbolo de clausura, entonces,

- Si el stack es vacío la expresión no es balanceada.
- Sino, si el tope del stack no es el correspondiente símbolo de apertura, la expression tampoco es balanceada.
- En el caso contrario, borre el tope del stack y siga inspeccionando la lista.

Si la lista es vacía y el stack también lo es, entonces es una expresión correctamente balanceada. Si el stack no es vacío, la expresión no es balanceada.

Invocación de Procedimientos y Funciones

El algoritmo para chequear balance de símbolos sugiere una forma de implementar invocación de procedimientos y funciones.

La diferencia en este caso es que cuando un procedimiento es invocado todas las variables locales al proceso invocador deben ser salvadas por el sistema ya que, entre otras razones, el procedimiento invocado puede sobreescribir las variables del invocador.

Cuando se invoca a un subprograma la siguiente información debe ser guardada:

Stack de frames

- valores de los registros correspondientes a variables del subprograma invocador;
- dirección de retorno, es decir, el lugar al que se transfiere el control una vez que el subprograma invocado termina de ejecutarse.

Todo este proceso puede claramente implementarse usando un stack. Cuando un subprograma es invocado , la información arriba descripta es salvada en una estructura (frame) y guardada en el tope del stack.

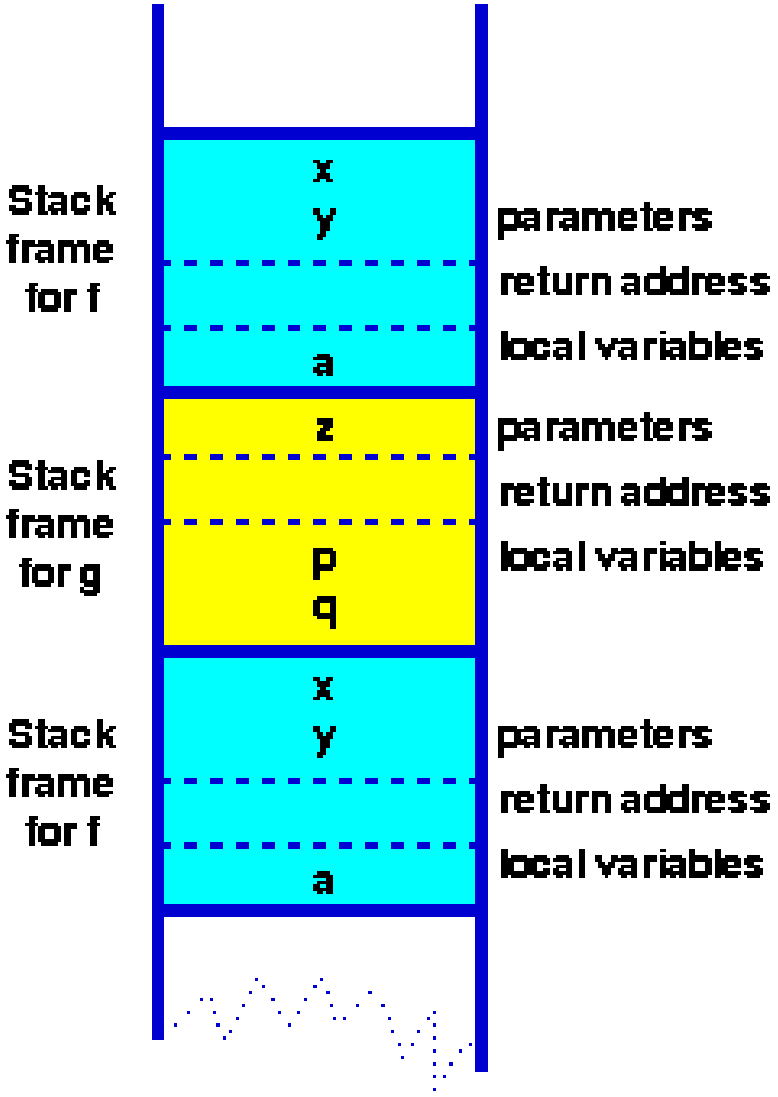
Flujo de control y puntos de retorno

Luego, el control se transfiere al subprograma invocado, el que puede libremente hacer uso de los registros para almacenar su información local. Si este subprograma a su vez invoca a otro, se repite este proceso.

Cuando el subprograma invocado desea retornar el control se fija en el frame que se encuentra en el tope del stack, recompone los registros y luego transfiere el control a la dirección de retorno que fue almacenada.

Stack de frames

```
function f (int x, int y) {
int a;
    if ( term_cond ) return ...;
    a = .....;
    return g(a);
    }

function g(int z) {
    int p,q;
    p = ...; q = ...;
    return f(p,q);
    }
```



Notar como los entornos de f y g (sus parámetros y variables locales) se encuentran en el stack de frames. Cuando f es llamada una segunda vez desde g, se crea un nuevo frame para la segunda invocación.

Flujo de control y puntos de retorno

Luego, el control se transfiere al subprograma invocado, el que puede libremente hacer uso de los registros para almacenar su información local. Si este subprograma a su vez invoca a otro, se repite este proceso.

Cuando el subprograma invocado desea retornar el control se fija en el frame que se encuentra en el tope del stack, recompone los registros y luego transfiere el control a la dirección de retorno que fue almacenada.

Ejemplos simples de uso

```
#include <stdio.h>
#include "pila.h"

void imprimirVacia (Pila pila) {
   if (esVaciaPila (pila)) {
        printf ("La pila está vacía.\n");
   } else {
        printf ("La pila no está vacía.\n");
   }
}

void imprimirLlena( Pila pila) {
   if (esLlenaPila (pila)) {
        printf ("La pila está llena.\n");
   } else {
        printf ("La pila no está llena.\n");
   }
}
```

Ejemplos simples de uso de Pilas (cont.)

```
int main () {
   Pila pila;
   pila = crearPila(5);

   imprimirVacia (pila);
   imprimirLlena (pila);

   for (int i = 0; i < 6; i++) {
        printf ("Apilo %d\n", i);
        apilar (i, pila);
        int top = cima (pila);
        printf ("La cima es %d\n", top);
   }

   imprimirVacia (pila);
   imprimirLlena (pila);
```

Ejemplos simples de uso de Pilas (cont.)

```
   while (! esVaciaPila (pila)) {
        int top = cima (pila);
        printf ("La cima es %d\n", top);
        desapilar (pila);
        printf ("Desapilo\n");
   }

   imprimirVacia (pila);
   imprimirLlena (pila);

   destruirPila(pila);

   return 0;
}
```

Ejercicios sobre el TAD Pila

1. Especifique el TAD Pila no acotada de elementos de un tipo T.
2. Implemente el TAD Pila no acotada usando una lista de memoria dinámica (como estructura de datos).
3. Implemente el pseudocódigo de la aplicación de balanceo de símbolos usando el TAD Pila.

Especificación del TAD Pila no acotada de elementos de tipo T

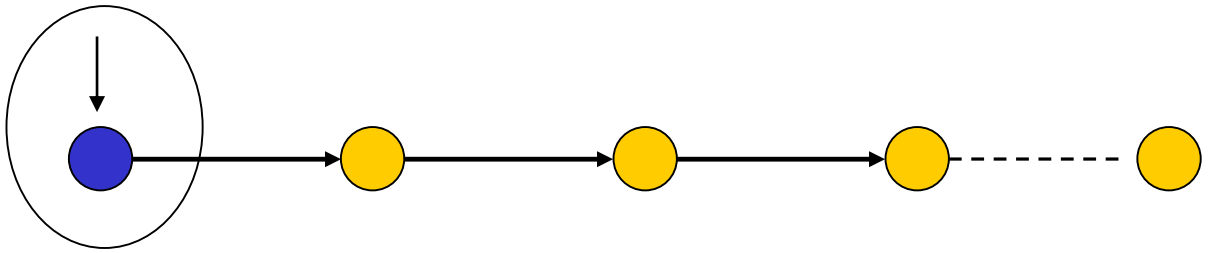
```
#ifndef _PILA_H
#define _PILA_H

struct RepresentacionPila;
typedef RepresentacionPila * Pila;

// CONSTRUCTORAS
Pila crearPila ();
// Devuelve la pila vacía.

void apilar (T i, Pila &p);
/* inserta i en la cima de p */

// SELECTORAS
T cima (Pila p);
/* Devuelve la cima de p.
   Precondicion: ! esVaciaPila(p). */

void desapilar (Pila &p);
/* Remueve la cima de p.
   Precondicion: ! esVaciaPila(p). */
```

Especificación del TAD Pila no acotada de elementos de tipo T

```
// PREDICADOS
bool esVaciaPila (Pila p);
/* Devuelve true si y sólo si p es vacia. */

// ADICIONAL
int cantidadPila (Pila p);
/* Devuelve la cantidad de elementos en la pila p. */


// DESTRUCTORA
void destruirPila (Pila &p);
/* Libera toda la memoria ocupada por p. */


#endif /* _PILA_H */
```

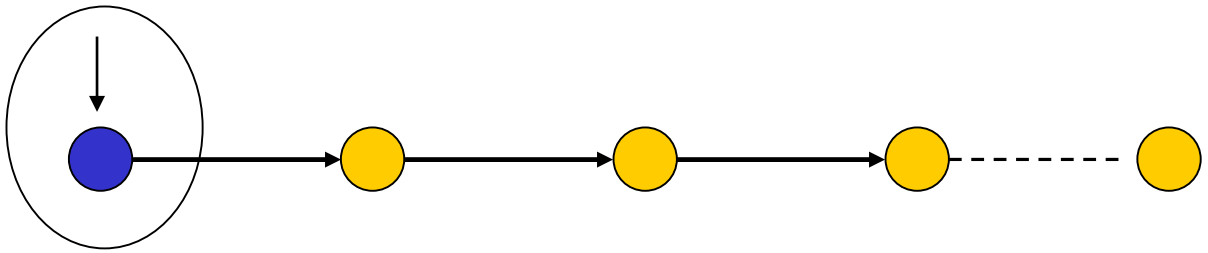
Implementación con una lista de memoria dinámica de una Pila no acotada de elementos de tipo T

```
#include <stddef.h>
#include <assert.h>

#include "pila.h"

struct nodoPila{
   T dato;
   nodoPila * sig;
}

struct representacionPila{
   nodoPila * pila;
   int cantidad;
}
```



Implementación (cont.)

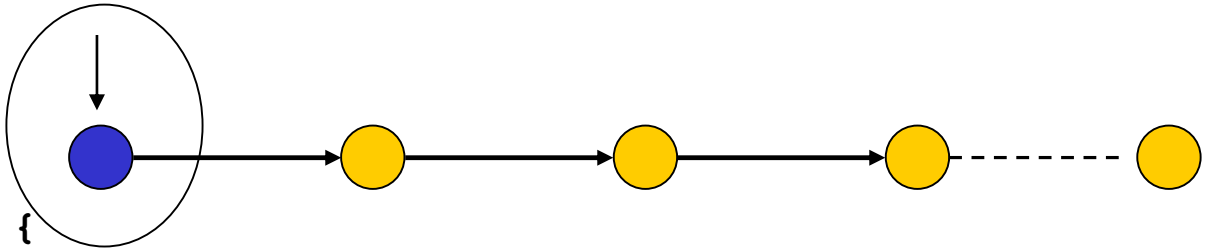
```
// CONSTRUCTORAS
Pila crearPila () {
   Pila p = new representacionPila;
   p->pila = NULL;
   p->cantidad = 0;
   return p;
}
```



```
void apilar (T i, Pila &p) {
   nodoPila * nuevo = new nodoPila;
   nuevo->dato = i;
   nuevo->sig = p->pila;
   p->pila = nuevo;
   p->cantidad++;
}
```

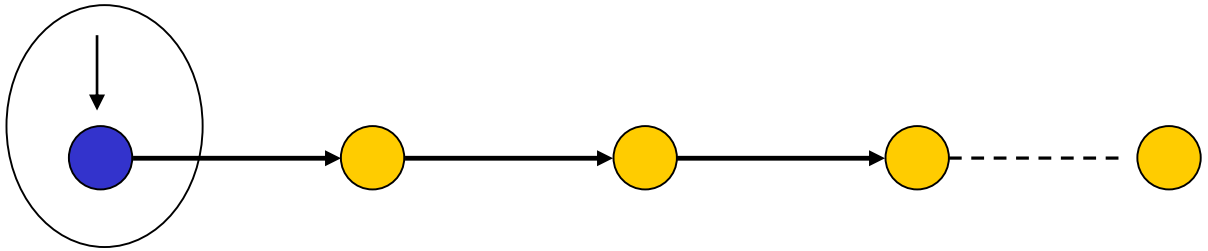
Implementación (cont.)

```
// SELECTORAS
T cima (Pila p) {
   assert (!esVaciaPila());
   return p->pila->dato;
}

void desapilar (Pila &p) {
   assert (!esVaciaPila());
   nodoPila * aBorrar = p->pila;
   p->pila = p->pila->sig;
   delete aBorrar;
   p->cantidad--;
}

// ADICIONAL
int cantidadPila (Pila p) {
   return p->cantidad;
}
```



Implementación (cont.)

```
// PREDICADOS
bool esVaciaPila (Pila p) {
   return (cantidadPila(p)==0);
}
```



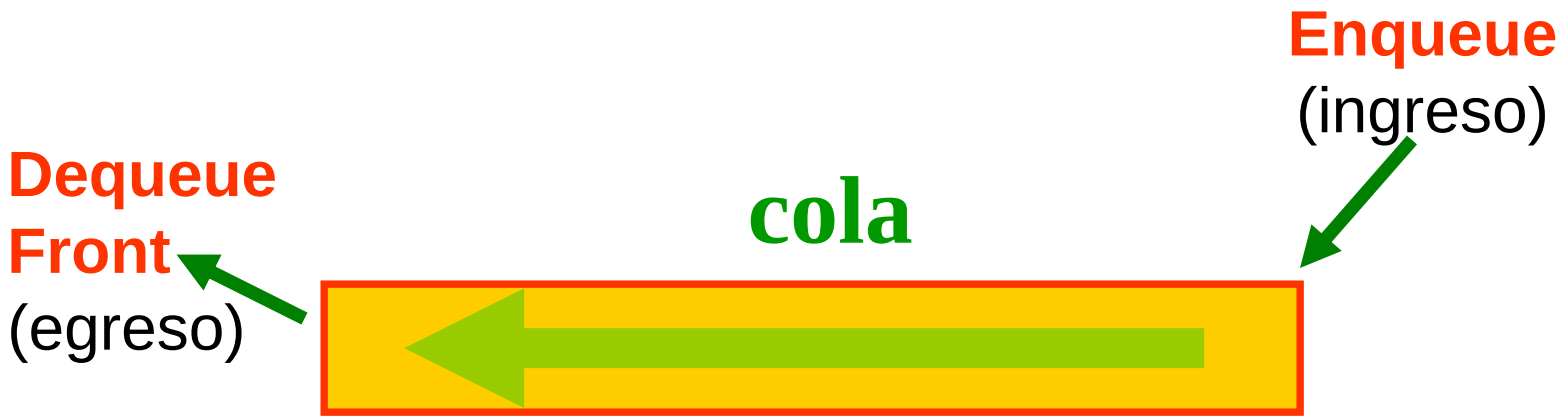
```
// DESTRUCTORA
void destruirPila (Pila &p) {
   while (!esVaciaPila(p)){
        desapilar(p);
   }
   delete p; // nodo de la representación
}
```

TAD COLA (QUEUE)

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, **Empty (crearCola)**.
- **Enqueue (encolar)**, que inserta un elemento al final de la cola.
- **Front (frente)**, que retorna el elemento que se encuentra en el comienzo de la cola.
- **Dequeue (desencolar)**, que borra el primer elemento de la cola.
- **IsEmpty (esVaciaCola)**, que testea si la cola es vacía.
- **IsFull (esLlenaCola)**, que testea si la cola está llena (si es una cola acotada).
- Una operación destructora, para eliminar una cola y liberar la memoria que ésta ocupa.

NOTAS:

- *Enqueue* podría no especificar donde se hacen las inserciones y en este caso las operaciones selectoras deberían referirse al primer elemento ingresado (el más antiguo).
- Si se trata de una cola acotada, *Enqueue* tendría precondición.

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). ¿Cómo se procesan los procesos en espera de un procesador?

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.....

Implementaciones 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.

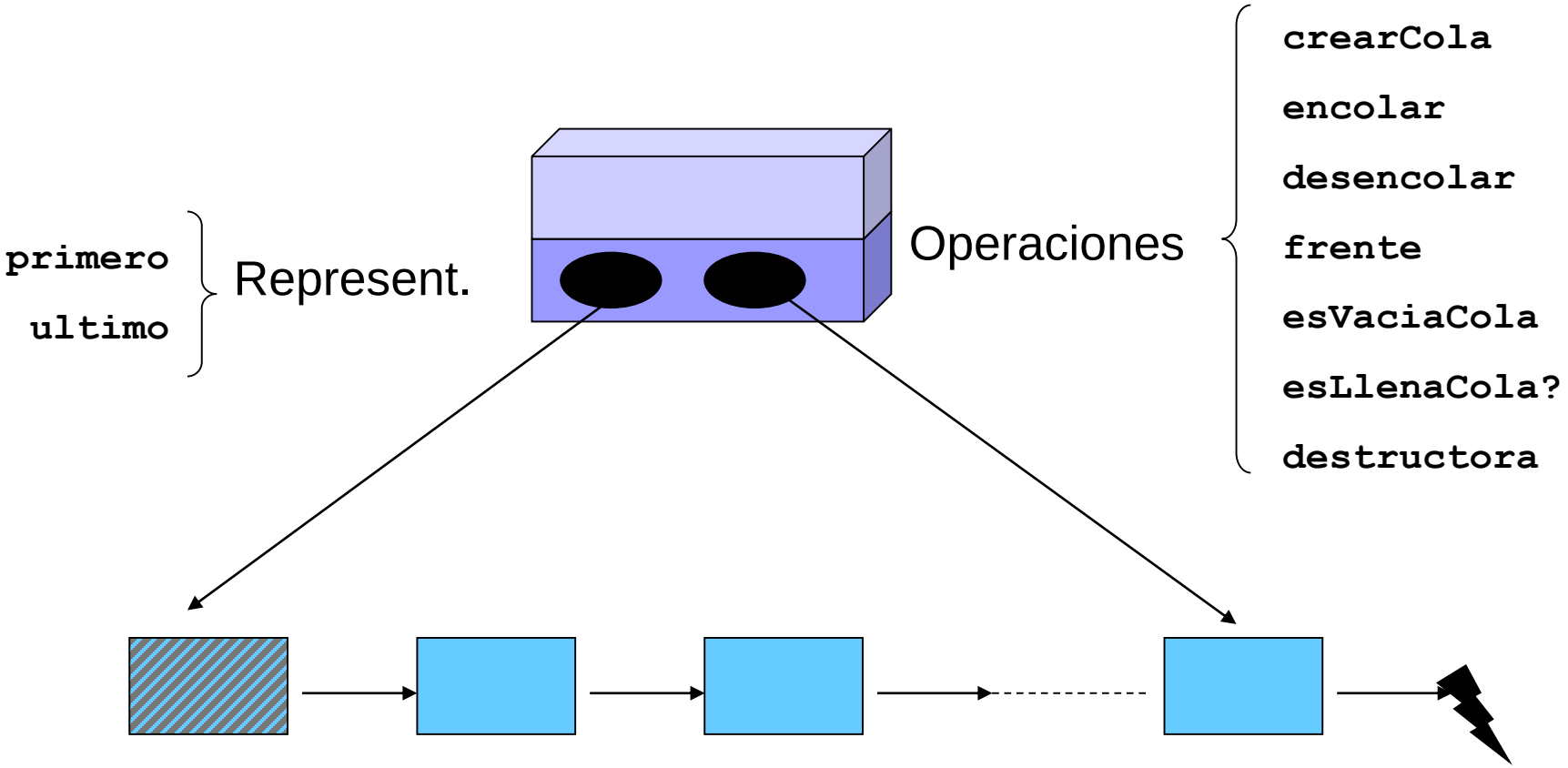
Implementación del TAD Queue (cont.)

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

Para colas, este puntero permitirá implementar **`Front`** y **`Dequeue`** en forma eficiente.

Se puede mantener además una celda (un nodo) "dummy“ (cabecera) como cabeza de la lista. El puntero **`ppio`** apuntará a esta celda. Esta convención permitiría hacer un manejo adecuado de la cola vacía. Si se hacen los controles adecuados (en los casos de borde), se puede omitir el uso de la celda dummy.

Implementación del TAD Queue (cont.)



Especificación e Implementación de un TAD Cola no acotada de elementos de un tipo T en C/C++

Especificación en C/C++ de un TAD
Cola no acotada de elementos de tipo T

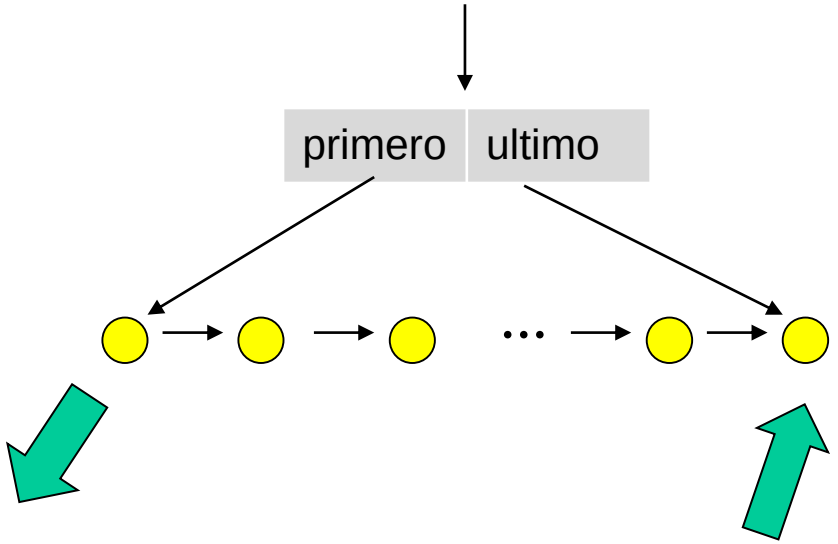
```
#ifndef _COLA_H
#define _COLA_H

struct RepresentacionCola;
typedef RepresentacionCola* Cola;

Cola crearCola ();
/* Devuelve la cola vacia. */

void encolar (T t, Cola &c);
/* Agrega el elemento t al final de c. */

bool esVaciaCola (Cola c);
/* Devuelve 'true' si c es vacia, 'false' en otro caso. */
```

Especificación en C/C++ de un TAD Cola no acotada de elementos de tipo T

```
T frente (Cola c);
/* Devuelve el primer elemento de c.
   Precondicion: ! esVaciaCola(c). */

void desencolar (Cola &c);
/* Remueve el primer elemento de c.
   Precondicion: ! esVaciaCola(c). */

void destruirCola (Cola &c);
/* Libera toda la memoria ocupada por c. */

#endif /* _COLA_H */
```

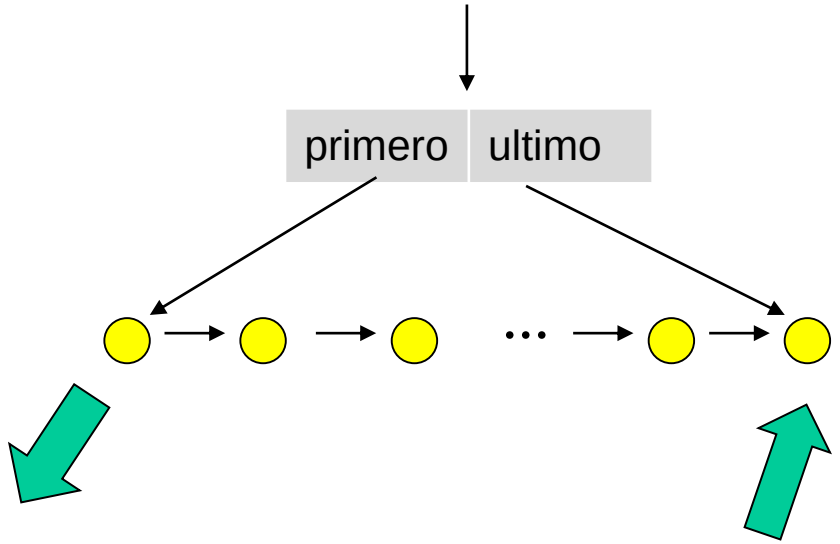
Implementación en C/C++ de un TAD Cola no acotada de elementos de tipo T

```
#include "Cola.h"
#include <stddef.h>
#include <assert.h>

struct Nodo
{
    T valor;
    Nodo * sig;
}

struct RepresentacionCola
{
    Nodo *primero, *ultimo;
}

Cola crearCola ()
{
    Cola c = new RepresentacionCola;
    c->primero = c->ultimo = NULL;
    return c;
}
```



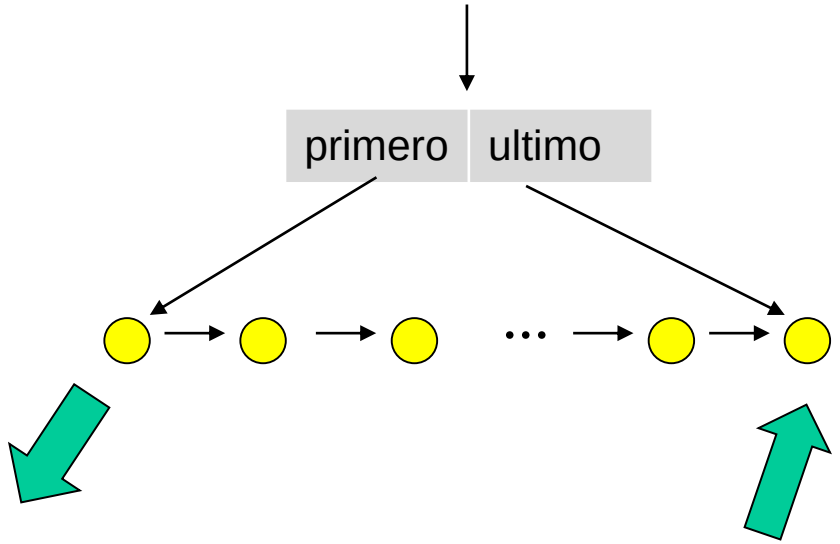
Implementación en C/C++ de un TAD Cola no acotada de elementos de tipo T

```
void encolar (T t, Cola &c)
{
    Nodo *nuevo = new Nodo;
    nuevo->valor = t;
    nuevo->sig = NULL;
    if (c->primero == NULL) c->primero = nuevo;
       else c->ultimo->sig = nuevo;
    c->ultimo = nuevo;
}

bool esVaciaCola (Cola c)
{
    return (c->primero == NULL);
}

T frente (Cola c)
{
    assert(!esVaciaCola(c));
    return c->primero->valor;
}
```



Implementación en C/C++ de un TAD Cola no acotada de elementos de tipo T

```
void desencolar (Cola &c)
{
    assert(c->primero != NULL);
    Nodo *aBorrar = c->primero;
    c->primero = c->primero->sig;
    if (c->primero == NULL) c->ultimo = NULL;
    delete aBorrar;
}

void destruirCola (Cola &c)
{
   while (!esVaciaCola(c))
    {
        desencolar(c);
    }
    delete c;
}
```



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.

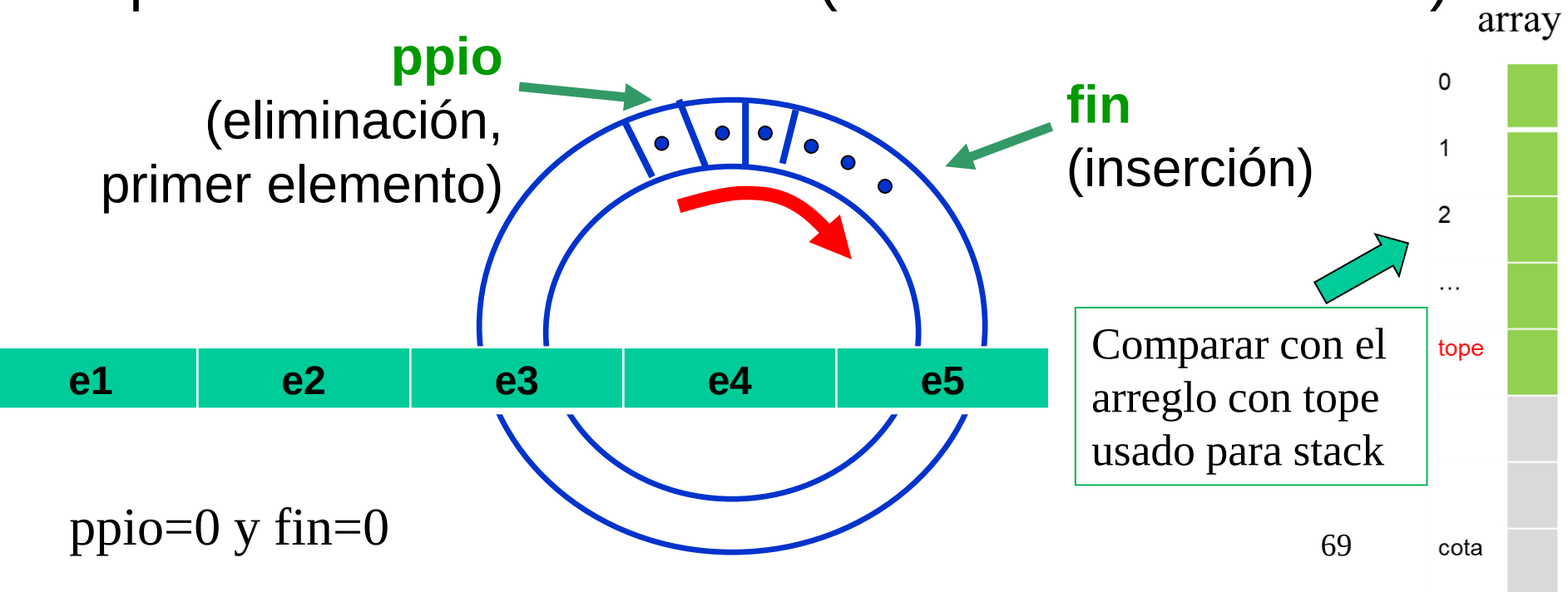
Pero dar de baja al primer elemento de la cola requiere desplazar todos los elementos una posición en el arreglo.

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).



ppio=0 y fin=0

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 en la representación que lleve el tamaño de la cola (contador).
- Prevenir que la cola no ocupe todo el arreglo.
- ¿Alguna otra?

¿Cuál te parece la solución más clara/simple?

Ejercicio: Especifique e Implemente el TAD Cola acotada usando arreglos circulares.

Especificación del TAD Cola acotada en C/C++

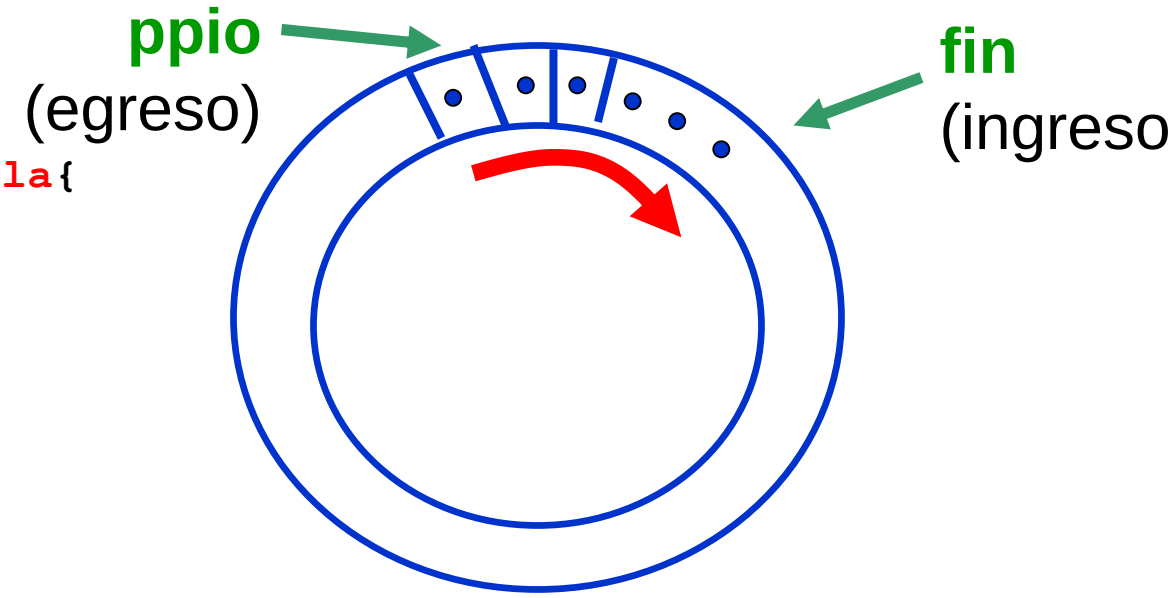
```
#ifndef _COLA_H
#define _COLA_H

struct RepresentacionCola;
typedef RepresentacionCola * Cola;

// CONSTRUCTORAS
Cola crearCola (int cota);
// Devuelve una cola vacía que podrá contener hasta cota elementos.

void encolar (T i, Cola &c);
/* Inserta i en la cola c
   Precondicion: !esLlenaCola(c). */

// SELECTORAS
T frente (Cola c);
/* Devuelve el primero de c en orden fifo (el primero que llegó).
   Precondicion: !esVaciaCola(c). */

void desencolar (Cola &c);
/* Remueve el elemento de c en orden fifo (el primero que llegó).
   Precondicion: !esVaciaCola(c). */
```

Especificación del TAD Cola acotada en C/C++

```
// PREDICADOS
bool esVaciaCola (Cola c);
/* Devuelve true si y sólo si c es vacia. */

bool esLlenaCola (Cola c);
/* Devuelve 'true' si y sólo si c tiene cota elementos, donde cota
   es el valor del parámetro pasado en crearCola. */

// DESTRUCTORA
void destruirCola (Cola &c);
/* Libera toda la memoria ocupada por c. */


#endif /* _COLA_H */
```

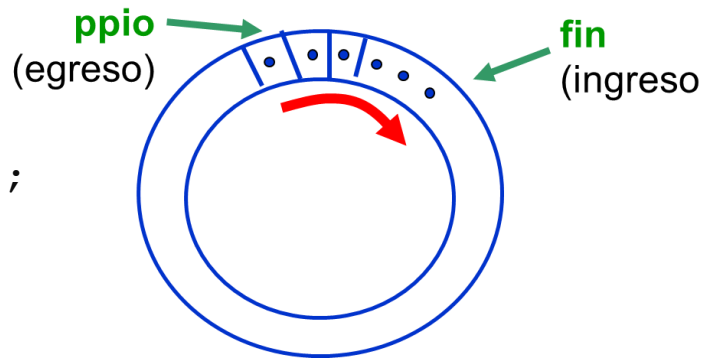
Implementación con un vector de una Cola acotada de enteros en C/C++

```
#include <stddef.h>
#include <assert.h>

#include "cola.h"

struct RepresentacionCola{
   T* array;
   int ppio;
   int fin;
   int cantidad;
   int cota;
};

Cola crearCola (int cota) {
   Cola c = new RepresentacionCola();
   c->ppio = c->fin = c->cantidad = 0;
   c->array = new T[cota];
   c->cota = cota;
   return c;
}
```



Implementación con un vector de una Cola acotada de enteros en C/C++

```
void encolar (T i, Cola &c){
   assert(!esLlenaCola(c));
   c->array[c->fin] = i;
   c->fin++; if (c->fin==c->cota) c->fin = 0;
   c->cantidad++;
}

T frente (Cola c){
   assert(!esVaciaCola(c));
   return c->array[c->ppio];
}

void desencolar (Cola &c){
   assert(!esVaciaCola(c));
   c->ppio++; if (c->ppio==c->cota) c->ppio = 0;
   c->cantidad--;
}

bool esVaciaCola (Cola c){ return (c->cantidad == 0); }
bool esLlenaCola (Cola c){ return (c->cantidad == c->cota); }

void destruirCola (Cola &c){
   delete [] c->array; delete c;
}
```



Implementación en C/C++ de un TAD Cola acotada de elementos de tipo T

```
#include "Cola.h"
#include <stddef.h>
#include <assert.h>

struct Nodo
{
    T valor;
    Nodo * sig;
};

struct RepresentacionCola
{
   Nodo *primero, *ultimo;
   int cota; int cantidad;
};

Cola crearCola (int cota)
{
    Cola c = new RepresentacionCola;
    c->primero = c->ultimo = NULL;
    c->cantidad = 0; c->cota = cota;
    return c;
}
```

Adaptación de la implementación realizada para la cola no acotada, usando listas de memoria dinámica

Implementación en C/C++ de un TAD Cola no acotada de elementos de tipo T

```
void encolar (T t, Cola &c)
{   assert(!esLlenaCola(c));
    Nodo *nuevo = new Nodo;
    nuevo->valor = t;
    nuevo->sig = NULL;
    if (c->primero == NULL) c->primero = nuevo;
        else c->ultimo->sig = nuevo;
    c->ultimo = nuevo;
    c->cantidad++;
}

bool esVaciaCola (Cola c)
{
    return (c->cantidad == 0);
}

bool esLlenaCola (Cola c)
{
    return (c->cantidad == c->cota);
}

T frente (Cola c)
{
    assert(!esVaciaCola(c));
    return c->primero->valor;
}
```

Implementación en C/C++ de un TAD Cola no acotada de elementos de tipo T

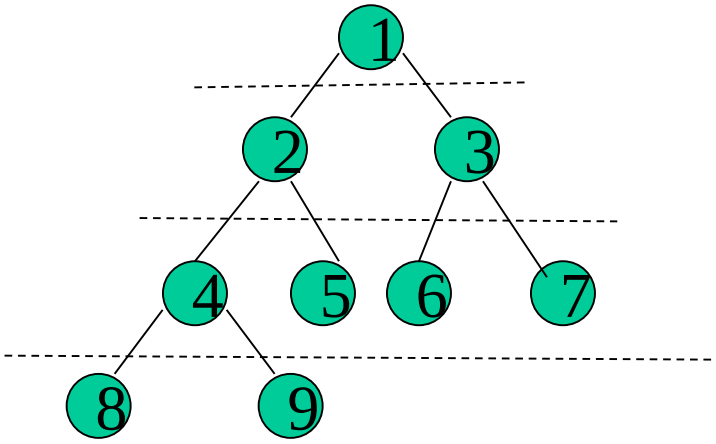
```
void desencolar (Cola &c)
{
    assert(c->primero != NULL);
    Nodo *aBorrar = c->primero;
    c->primero = c->primero->sig;
    if (c->primero == NULL) c->ultimo = NULL;
    delete aBorrar;
    c->cantidad--;
}

void destruirCola (Cola &c)
{
   while (!esVaciaCola(c))
    {
        desencolar(c);
    }
    delete c;
}
```

Aplicación del TAD Cola (Queue): Recorrido por niveles

¿Qué es un recorrido por niveles de un árbol? ¿Qué aplicaciones tiene?

¿Cómo se puede recorrer un árbol binario por niveles?

¿Es posible resolver este problema en O(*n*), siendo *n* la cantidad de nodos del árbol?

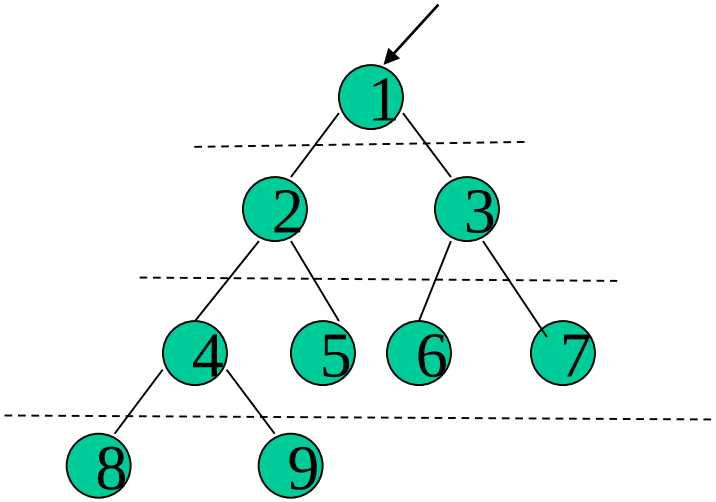


Imprimir un nivel de un árbol binario

```
void imprimirNivelAB (NodoAB* t, in i){
    if (t!=NULL && i>0){
        if (i>1){
            imprimirNivelAB(t->izq, i-1);
            imprimirNivelAB(t->der, i-1);
        }
        else{ // i==1
            cout << t-dato;
        }
    }
}
```

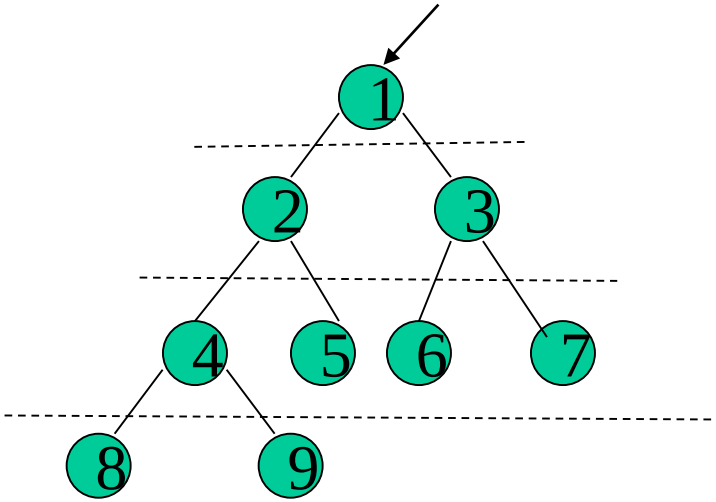
NOTA: Calcular la altura del árbol (h) y luego usar imprimirNivelAB para imprimir cada nivel entre 1 y h genera un algoritmo de orden $O(n^2)$…

Aplicación del TAD Cola (Queue): Recorrido por niveles de un árbol binario



Cola de árboles (de punteros)

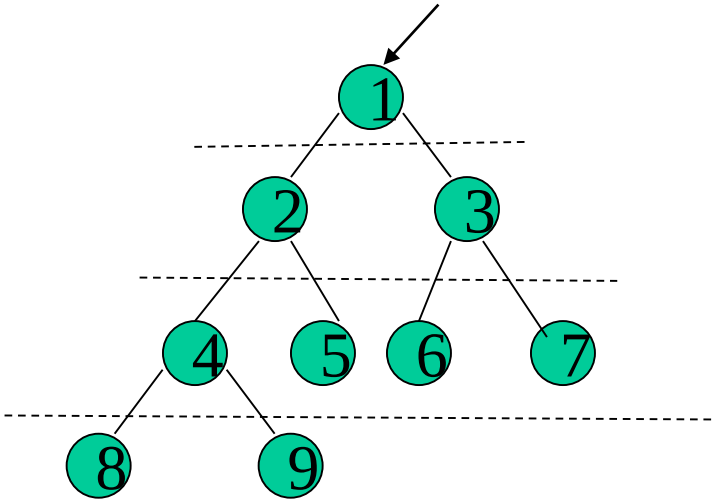
Aplicación del TAD Cola (Queue): Recorrido por niveles de un árbol binario



Cola de árboles (de punteros)

Si imprime: 1

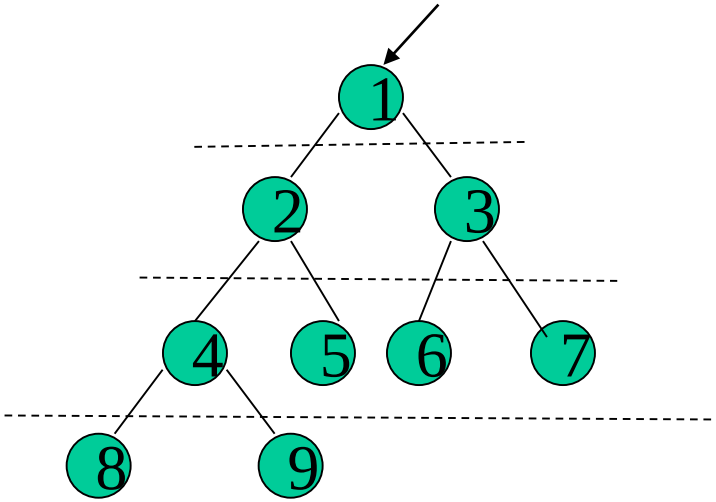
Aplicación del TAD Cola (Queue): Recorrido por niveles de un árbol binario

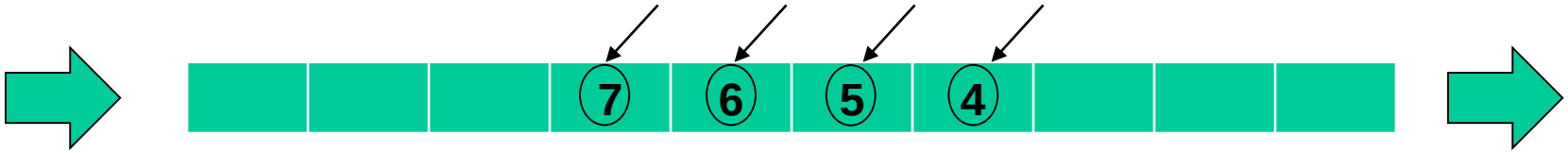


Cola de árboles (de punteros)

Si imprime: 1, 2

Aplicación del TAD Cola (Queue): Recorrido por niveles de un árbol binario

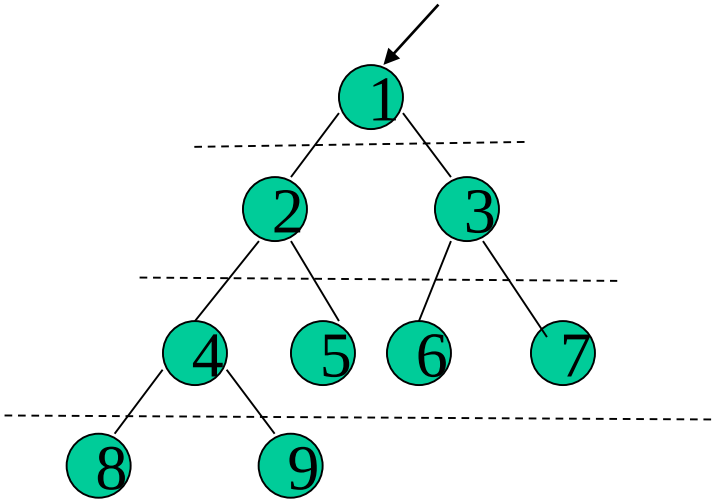


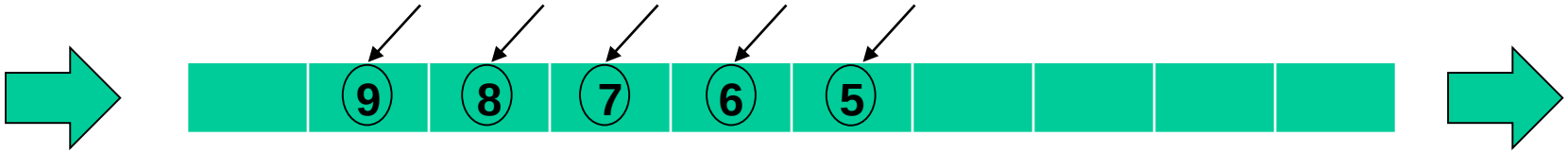


Cola de árboles (de punteros)

Si imprime: 1, 2, 3

Aplicación del TAD Cola (Queue): Recorrido por niveles de un árbol binario

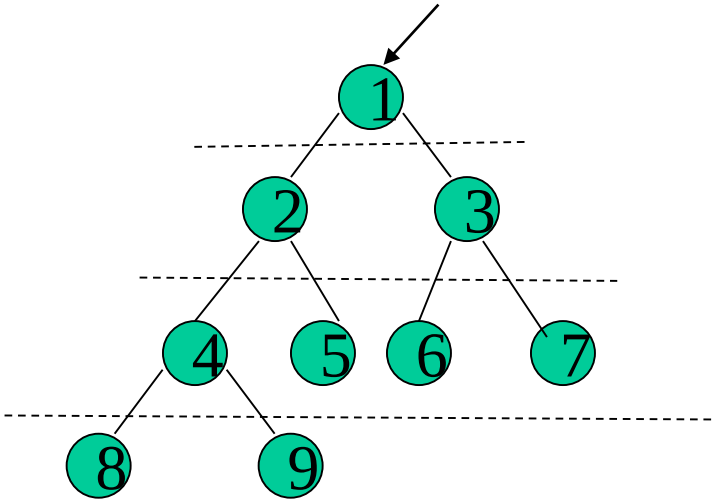


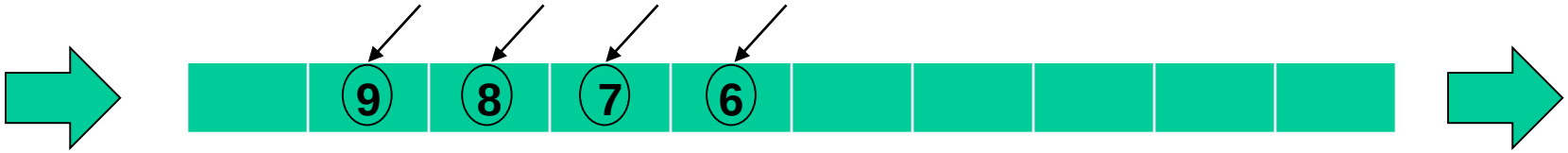


Cola de árboles (de punteros)

Si imprime: 1, 2, 3, 4

Aplicación del TAD Cola (Queue): Recorrido por niveles de un árbol binario

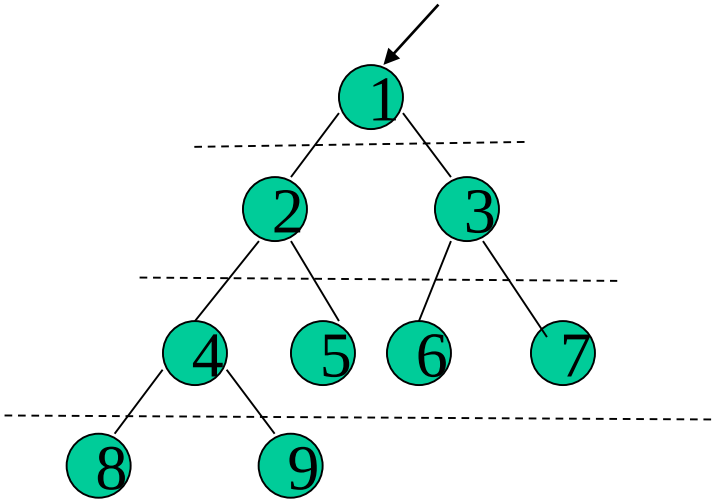




Cola de árboles (de punteros)

Si imprime: 1, 2, 3, 4, 5

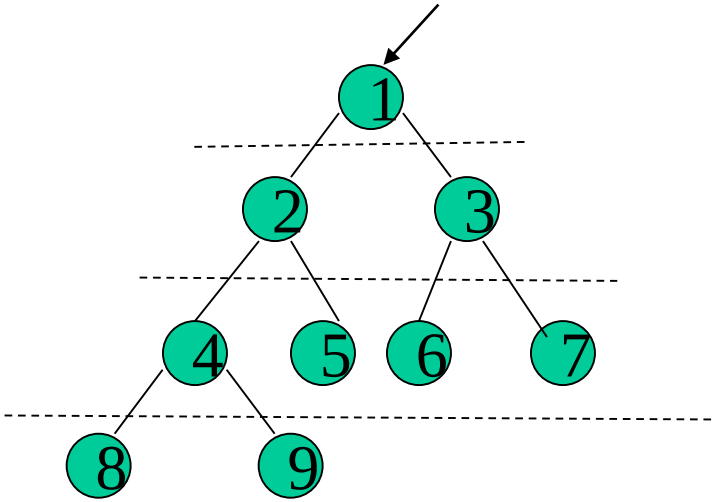
Aplicación del TAD Cola (Queue): Recorrido por niveles de un árbol binario



Cola de árboles (de punteros)

Si imprime: 1, 2, 3, 4, 5, 6

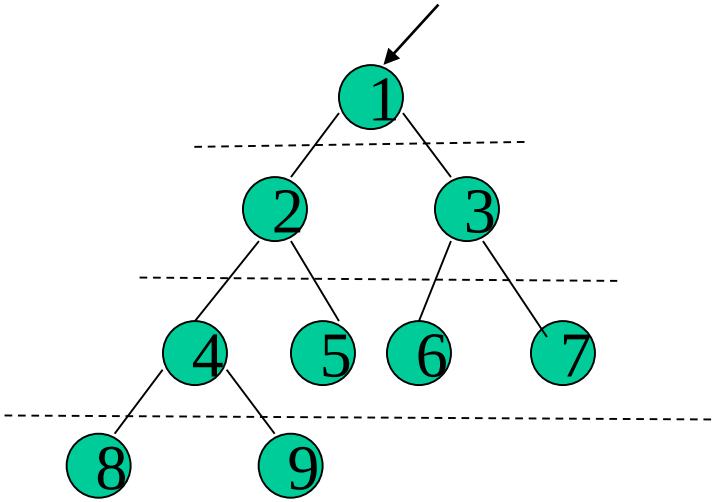
Aplicación del TAD Cola (Queue): Recorrido por niveles de un árbol binario



Cola de árboles (de punteros)

Si imprime: 1, 2, 3, 4, 5, 6, 7

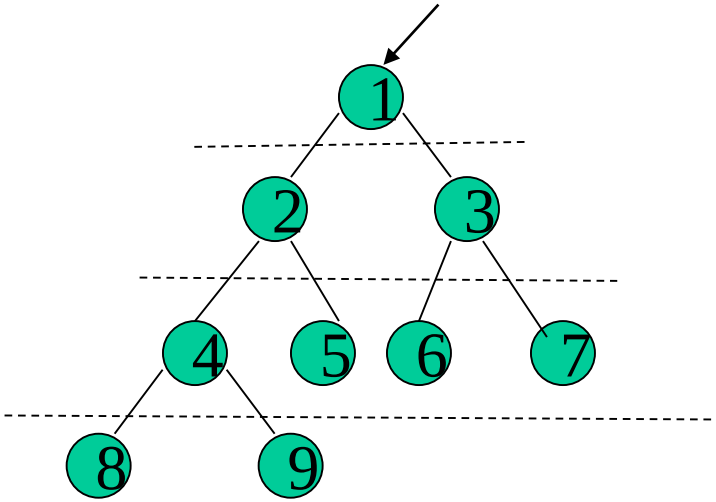
Aplicación del TAD Cola (Queue): Recorrido por niveles de un árbol binario



Cola de árboles (de punteros)

Si imprime: 1, 2, 3, 4, 5, 6, 7, 8

Aplicación del TAD Cola (Queue): Recorrido por niveles de un árbol binario



Cola de árboles (de punteros)

Si imprime: 1, 2, 3, 4, 5, 6, 7, 8, 9

FIN

Aplicación del TAD Cola (Queue): Recorrido por niveles de un árbol binario

Imprimir un árbol binario *a* por niveles:

- Si *a* no es vacío, encolar *a* en una cola *c* inicialmente vacía de árboles binarios.
- Mientras la cola *c* no sea vacía:
  - Obtener el primer (en orden *fifo*) árbol de *c;*
  - Imprimir el dato de su ráiz;
  - Para sus subárboles izquierdo y derecho, si no son vacíos entonces encolarlos en *c;*
  - Desencolar de *c;*

Escribir el código en C++ para un árbol de enteros.

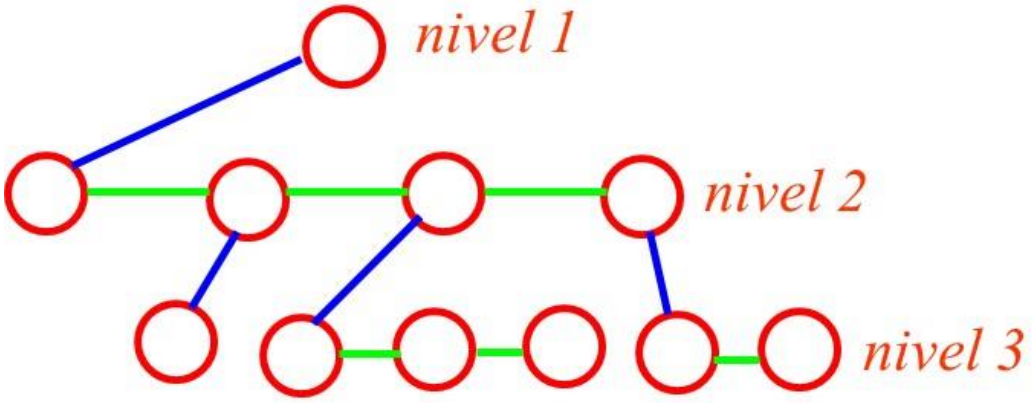
Aplicación del TAD Cola (Queue): Recorrido por niveles de un árbol binario

```
void imprimirABxNiveles(NodoAB* t){
\\ usamos una cola de elementos de tipo NodoAB* (dato,izq,der)
    ColaAB c = crearColaAB();
    If (t != NULL) encolarColaAB(t, c);
    while (!esVaciaColaAB(c)){
            t = frenteColaAB(c);
            cout << t->dato;
            If (t->izq != NULL) encolarColaAB(t->izq, c);
            If (t->der != NULL) encolarColaAB(t->der, c);
            desencolarColaAB(c);
    }
    destruirColaAB(c);
}
```

Aplicación del TAD Cola (Queue): Recorrido por niveles de árboles generales

¿Cómo se puede recorrer un árbol general (finitario) por niveles? ¿Es posible hacerlo en O(n), siendo n la cantidad de nodos del árbol?

Generalice la solución realizada para árboles binarios, considerando árboles generales representados como binarios con la semántica: primer hijo – siguiente hermano.



Aplicación del TAD Cola (Queue): Recorrido por niveles de un árbol general

```
void imprimirAGxNiveles(NodoAG* t){
    \\ usamos una cola de elementos de tipo NodoAG* (dato,pH,sH)
    ColaAG c = crearColaAG();
    while (t != NULL) { enconlarColaAG(t); t = t->sH; }
    // Contempla el caso en que t es un bosque y no solo un árbol.
    while (!esVaciaColaAG(c)){
        t = frenteColaAG(c);
        cout << t->dato;
        t = t->pH;
        while (t != NULL){
            encolarColaAG(t,c);
            t = t->sH;
        }
        desencolarColaAG(c);
    }
    destruirColaAG(c);
}
```

Se encolan los hijos del nodo cuyo dato se imprimió.

Bibliografía

– 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.

– Como Programar en C/C++ (o versión sólo C++)

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