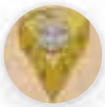


QuesoViejo_
www.wuolah.com/student/QuesoViejo_

Ejercicios Resueltos.pdf

Ejercicios Resueltos

1° Estructura de Datos y Algoritmos

Grado en Ingeniería Informática

Escuela Politécnica Superior
UC3M - Universidad Carlos III de Madrid

TADS RESUELTOS

2x1

carné universitario

FOSTER'S HOLLYWOOD

*Consulta las condiciones de la promoción en fostershollywood2x1universitario.com

TADS EXAMENES

2. Se tiene un TAD Tren con una secuencia de elementos tipo vagón. Si el tren no está vacío entonces hay un vagón activo que puede ser manipulado por un robot. Hacer la especificación de las siguientes operaciones:
   a. Construir un tren vacío.
   b. Desplazar tren a la izquierda (el vagón activo pasa a ser el de la derecha, si no hay vagón a la derecha no se hace nada).
   c. Desplazar tren a la derecha (el vagón activo pasa a ser el de la izquierda, si no hay vagón a la izquierda no se hace nada).
   d. Eliminar vagón activo: El vagón activo pasa a ser el de la derecha, si no hay vagón a la derecha, pasa a ser el de la izquierda, si tampoco hay, el tren está vacío.
   e. Observar vagón activo: Devuelve cuál es el vagón activo.
   f. Comprobar si el tren está vacío.

   Nota 1: Prohibido hacer doble enlazada con coste independiente.
   Nota 2: Definir los tipos usados y los prototipos de los TAD conocidos.

Definición: Secuencias de elementos de tipo vagón con un elemento que será el vagón activo. La secuencia empieza con el elemento primero y acaba con el elemento último

Operaciones

Tren ();

Postcondiciones: Crea un tren vacío

void DespIzq();

Precondiciones: El tren no está vacío

Postcondiciones: Si existe otro vagón a la derecha del VagonActivo, VagonActivo pasa a ser el de la derecha. En caso contrario no sucede nada.

void DespDcha ();

Precondiciones: El tren no está vacío

Postcondiciones: Si existe un vagón a la izquierda del vagonActivo, VagonActivo pasa a ser el vagón de la izquierda). En caso contrario no sucede nada.

void EliminarActivo();

Precondiciones: El tren no está vacío

Postcondiciones: El vagón activo pasa a ser el de la derecha. En caso de que no haya ninguno, el de la izquierda. Si no hay vagón ni a la derecha ni a la izquierda (es decir, solo había un vagón), se elimina el vagón y pasa a estar vacío.

const vagón& ObservarActivo () const ;

Precondiciones: No es un tren vacío

Postcondiciones: Devuelve el valor del vagón activo.

bool vacio () const ;

Postcondiciones: Devuelve true si es un tren vacío o false en su defecto.

*El tipo vagón de momento supongo que lo dan hecho

Prototipos conocidos del TAD Pila

* Escribir pre y postcond

```
bool Pila::vacia() const;
```

Postcondición: Devuelve true si la pila está vacía o false de lo contrario

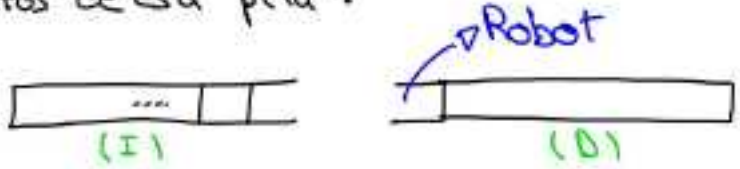
```
template <typename T>
const T& Pila::tope() const;
```

Precondición: La pila no está vacía

Postcondición: Devuelve el elemento en el tope de la pila (el que menos tiempo lleva en la pila.

Implementación:

/* La estructura de datos subyacente que se usa van a ser dos pilas de vagones de forma que el tope de la segunda pila sea el vagón activo, el resto de elementos de esa pila:



*/

// Tren.h

```
#ifndef Trenecito
#define Trenecito
#include <cassert>
class Tren {
public:
```

```
    Tren();   // Constructor
    void DespIzq();
    void DespDcha();
    void EliminarActivo
    const vagon& ObservarActivo() const;
    bool vacio() const;
    ~Tren();
private:
    Pila<vagon> I ;// Pila dinámica (la de la izquierda)
    Pila<vagon> D; // Pila dinámica (la de la derecha)
    //El vagón activo será el tope de la pila D
};
```

```
// Crea un tren vacío
Tren::Tren(){}

void Tren::DespIzq(){
  assert (!vacio());
  vagon aux= D.tope();
  D.pop();
  if(!D.vacia)}
    I.push(aux);
  }else} // si el tope es el último, se vuelve a poner
         // donde estaba.
```

Este método pertenece a la clase Tren y el objeto que lo llama es el mismo que llama a DespIzq. Esta llamada es equivalente a this.vacio()

```
    D.push (tope);
   {
}

void Tren::DespDcha()}
  assert (!vacio());   // Para verificar la precondición.
                       // No es obligatorio ponerlo.

  if(!I.vacia)}  // con esto ya se comprueba que hay
                 // algún elemento a la izquierda del vagón activo
  vagon aux= I.tope();
  I.pop();
   D.push (aux);
  {
/* El enunciado dice que si no hay ningún vagón
activo a la izquierda, este método no hace nada, por lo
que poner un assert sería un error ya que detendría la
ejecución
*/
void Tren::EliminarActivo ()}
  assert (!D.vacia()); // Para verificar la precondicion.
                       // No es obligatorio ponerlo

  D.pop
  if( D.vacia())}
   if(!I.vacia())}
     D.push (I.tope());
     I.pop
  {
```

```
// Si no ha entrado en ninguno de los if es que solo
// había un vagón y lo he sacado, quedando el tren vacío.

const vagon& Tren::ObservarActivo()const {
  assert(!D.vacia());
  return D.tope();
}

bool Tren::vacio()const {          // Este es el método con el que se comprobaban las precondiciones
  bool aux=(!I.vacia() && !D.vacia());
  return aux;
}
/* En principio no hay manera de que en un tren que
no esté vacío, no haya un VagónActivo, pero por si acaso
hago la comprobación en las dos pilas.
*/

Tren::~Tren() {
  I.~Pila();
  D.~Pila();
}
```

UCA Universidad de Cádiz

Análisis de Algoritmos y Estructuras de Datos

Grado en Ingeniería Informática

4 de Febrero de 2016

La dirección de un hospital quiere informatizar su consultorio médico que está en constante crecimiento por medio de un sistema que permita realizar al menos las siguientes operaciones:

- Generar un consultorio vacío sin ninguna información.
- Dar de alta un nuevo médico.
- Dar de baja a un médico.
- Poner a un paciente en la lista de espera de un médico.
- Consultar el paciente a quién le toca el turno para ser atendido por un médico.
- Atender al paciente que le toque por parte de un médico.
- Comprobar si un médico determinado tiene o no pacientes en espera.

a) Realice la especificación del TAD.

b) Diseñe una estructura de datos adecuada para representar el TAD e implemente las operaciones anteriores.

Nota: Es absolutamente necesario definir todos los tipos de datos implicados en la resolución del ejercicio, así como los prototipos de las operaciones utilizadas de los TADs conocidos.

Especificación :

Un consultorio es una secuencia de médicos de longitud $n \geq 0$ de forma que cada uno tiene una cola de pacientes en espera. Si $n=0$ significa que el consultorio está vacío

Identificador : Sirve para a un médico u otro dentro del consultorio. Los médicos se ordenan de forma lineal: $C=(m_1, m_2 \ldots, m_n)$ según el identificador. Todos menos el primero tienen un predecesor y todos menos el último tienen un un sucesor

Paciente : Entidad que se asocia a los médicos para que les atiendan.

Operaciones

Consultorio ();

Postcondiciones: Crea un consultorio vacío

void Consultorio :: NuevoMédico ()

Postcondiciones: Añade un médico al consultorio

void Consultorio :: BajaMédico (identificador id)

Precondiciones: El médico id pertenece al consultorio

Postcondiciones: Elimina a ese médico del consultorio.

void PonerPaciente(identificador id, paciente pa)

Precondiciones: El médico id pertenece al consultorio

Postcondiciones: Añade al paciente pa a la lista de espera del médico id (se sitúa el último).

paciente ConsultarPaciente(identificador id) const ;

Precondiciones: El médico id pertenece al consultorio

Postcondiciones: Devuelve el paciente del médico id al que le toca el turno

void AtenderPaciente(identificador id)

Precondiciones: El médico id debe pertenecer al consultorio y tener algún paciente

Postcondiciones: El paciente del médico id es atendido y el turno le pasa al siguiente.

bool SinEspera(identificador id) const

Precondiciones: El médico id pertenece al consultorio

Postcondiciones: Devuelve true si el médico id no tiene ningún paciente en espera o false en caso contrario

Métodos utilizados de otras TADS:

```
template <typename T> Cola<T>::Cola();
template<typename T> void Cola<T>::push(const T&x);
template <typename T> void Cola<T>::pop()
template <typename T> const T& Cola<T>::frente();
template <typename T> bool Cola<T>::vacia();
template <typename T> Cola<T>::~Cola();

template <typename N> Lista<N>::Lista();
template <typename N>
void Lista<N>::insertar(const N& x, posicion p);

template <typename N> void Lista<N>::eliminar(posicion p);

template <typename N>
const N& Lista<N>::elemento(posicion p);

template <typename N> posicion Lista<N>::primera();

template<typename N> posicion Lista<N>::siguiente(posicion p);

template <typename N> Lista<N>::~Lista();
```

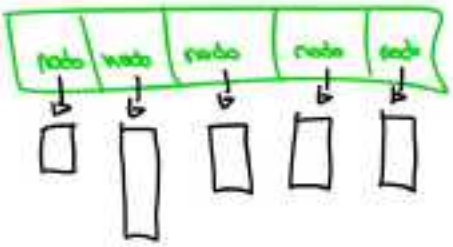
Implementación:

/* Los médicos se representan con colas donde se almacenan los pacientes mientras que el consultorio será una Lista de médicos, es decir, una lista de colas de pacientes.

Como el consultorio está "en continuo crecimiento" y no especifica que haya un límite de pacientes por médico, se usará la implementación dinámica del TAD Cola y la implementación dinámica simplemente enlazada con cabecera del TAD Lista

*/

```
class Consultorio{
 typedef Lista<medico>::posicion identificador;
public:
 struct paciente{
  int num;
 };
/* Como el ejercicio no pide ningún requisito para los pacientes, serán representados por un entero */

 Consultorio();    // Constructor
 void NuevoMedico();
 void BajaMédico(identificador id);
 void PonerPaciente(identificador id);
```



```
    const pacientel ConsultarPaciente(identificador id) const;
    void AtenderPaciente (identificador id);
    bool SinEspera (identificador id) const;
    ~Consultorio ();

  private:
    struct medico}
      Cola <paciente> col;
      medico () { } // Constructor de médico
    };
    Lista <nodo> C;
};
```

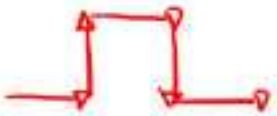
Implementación de los métodos:

```
Consultorio :: Consultorio () {} // Constructor vacío

void Consultorio :: NuevoMedico ()}
  C. insertar (new nodo(), C. primera());
{
// Los nuevos médicos los inserto al principio

void Consultorio :: BajaMedico (identificador id)}
    C. eliminar (id);
  {
```

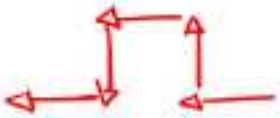
```
void Consultorio::PonerPaciente(identificador id,
                                paciente pa)}
  C.elemento(id).col.push(pa);
{

const paciente& Consultorio::ConsultarPaciente
                (identificador id)}

return C.elemento(id).col.frente();
}

void Consultorio::AtenderPaciente (identificador id)}

  C.elemento(id).col.pop();
}

bool Consultorio::SinEspera (identificador id)}

  return C.elemento(id).col.vacia();
{

Consultorio::~Consultorio (){
  identificador idini = C.primera();
  while (idini != C.fin()){ // Llamo al destructor de cada cola
    C.elemento(idini).col.~Cola();
    idini = C.siguiente(idini);
  }
  C.~Lista(); Ahora llamo al destructor de la lista
{
```

El TAD Símbolo se usa para representar símbolos trazados con líneas rectas. Un Símbolo es una sucesión de Trazos, un trazo es una línea recta que puede tener sentido a izquierda, derecha, arriba o abajo. El TAD símbolo tiene que permitir realizar las siguientes operaciones con un símbolo: crear un símbolo vacío, añadir un trazo al final de un símbolo y deshacer los últimos n trazos; también debe permitir realizar operaciones que devuelvan el símbolo simétrico respecto al eje X, Y ý XY. Por ejemplo:

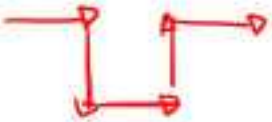
Símbolo



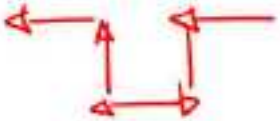
Simetría eje Y



Simetría Eje X



Simetría ejes XY



Especificación :

Un símbolo es una secuencia de Trazos de forma que se añaden o se eliminan trazos del final de esta secuencia. Si un símbolo no tiene ningún trazo, se denomina símbolo vacío.

Trazo: Elemento que forma parte del símbolo. Hay 4 tipos: Subir (↑), Bajar (↓), Derecha (→) e izquierda (←).

Operaciones

Símbolo ();

PostCondiciones: Crea un símbolo vacío.

void NuevoTrazo (Trazo t);

Precondiciones: t es un trazo válido

Postcondiciones: Añade t al final de la secuencia de trazos que ya formaban parte del símbolo. Si era un símbolo vacío pues t será el primer trazo.

void EliminarTrazos (int n);

Precondiciones: n > 0 y el símbolo tiene al menos n trazos

Postcondiciones: Elimina los últimos n trazos del símbolo.

Simbolo SimetriaX ();

Postcondicion: Devuelve un símbolo simétrico respecto al eje X

Simbolo SimetriaY ();

Postcondicion: Devuelve un símbolo simétrico respecto al eje Y

Simbolo SimetriaXY ();

Postcondicion: Devuelve un símbolo simétrico respecto a los ejes X e Y

~Simbolo();

Postcondiciones: Libera la memoria ocupada por la estructura interna del símbolo.

Implementación

```
#include <cassert>

enum Trazo {I, D, S, B};

class Simbolo {
public:
    Simbolo();  // Constructor de un símbolo vacío
    void NuevoTrazo(Trazo t);
    void EliminarTrazo(int n);

    const Simbolo& SimetriaX() const;
    const Simbolo& SimetriaY() const;
    const Simbolo& SimetriaXY() const;

    ~Simbolo();
private:
    Pila<Trazo> P;
    int nTrazos;
};
// Como no especifica un tamaño para el símbolo, se usara la
// implementación dinámica del TAD Pila

Simbolo::Simbolo(): nTrazos(0) {}

void Simbolo::NuevoTrazo(Trazo t){
    P.push(t);
    nTrazos++;
}
```

```
void Simbolo::EliminarTrazos (int n)
// assert(n>0 && nTrazos >= n)  No es obligatorio
// comprobar las precondiciones
while (n>0)
  P.pop();
  nTrazos --;
  n --;
}
{
```

```
const Simbolo & Simbolo::SimetriaX() const
  Pila<Trazo> aux(P);          // Con el constructor de copia
  Pila<Trazo> segunda;         // creo una pila auxiliar para
                               // no tocar la original.
  Simbolo sim; // simbolo vacío

  while (!aux.vacia())
    if(aux.tope() == Trazo::S)
     segunda.push(Trazo::B);
    else if(aux.tope() == Trazo::B)
     segunda.push(Trazo::S);
    else
     segunda.push(aux.tope());
    }
    aux.pop();
  {
```

```
    while ( ! segunda.vacia() )}   // Para dejar la pila al derecho
      sim.NuevoTrazo ( segunda.tope() ); //Inserto directamente
      segunda.pop();                     //en el objeto simbolo
    }                                    //que voy a devolver.
  return sim;
}


const  Simbolo &   Simbolo:: SimetriaY () const }
  Pila <Trazo > aux ( P );     //Con el constructor de copia
  Pila <Trazo > segunda ;      //creo una pila auxiliar para
                               //no tocar la original.
  Simbolo  sim ; //simbolo vacío

  while ( !aux.vacia ())}
    if ( aux.tope() == Trazo::D )}
     segunda.push (Trazo::I);
    else if(aux.tope() == Trazo::I)}
     segunda. push (Trazo::D);
    else}
     segunda . push( aux. tope() );
    }
    aux.pop() ;
  }
```

```
  while ( ! segunda.vacia() )}   // Para dejar la pila al derecho
    sim.NuevoTrazo ( segunda.tope() ); //Inserto directamente
    segunda.pop();                     //en el objeto simbolo
  }                                    //que voy a devolver.
 return sim;
}

const Simbolo &  Simbolo:: SimetriaXY() const }
  Pila <Trazo > aux ( P );     //Con el constructor de copia
  Pila <Trazo > segunda ;      // creo una pila auxiliar para
                               // no tocar la original.
  Simbolo sim ; //simbolo vacío

  while ( !aux.vacia ())}
    switch (aux.tope() )}
     case Trazo:: S  :}segunda.push(Trazo::B) ; break;{
     case Trazo:: B  :}segunda.push(Trazo::S);break; {
     case Trazo::I  : } segunda.push ( Trazo:: D); break; {
     case Trazo::D  : } segunda.push (Trazo :: I); break; {
    {
    aux.pop();
  {

  while (! segunda.vacia ())}
    Sim.NuevoTrazo ( segunda.tope() );
    Segunda.pop();
 {
```

```
    return Sim;
}

Simbolo :: ~Simbolo ()}
    P.~Pila();
{
```

Práctica (3 ptos)

Una empresa de muebles de cocina necesita un TAD para representar el conjunto de muebles colocados en la pared de una cocina. Una cocina se crea con una longitud positiva, y un mueble colocado en la pared se identifica con el par formado por su posición (distancia desde su lateral izquierdo al extremo izquierdo de la pared) y su anchura (la profundidad y altura no tienen interés, pues son iguales para todos los muebles).

El TAD debe soportar las siguientes operaciones:

- Crear una cocina vacía con una longitud dada.
- Determinar si un mueble de una cierta anchura puede colocarse en una posición dada.
- Añadir un mueble de una determinada longitud a una posición dada.
- Devolver el mueble i-ésimo de la cocina empezando a contar por la izquierda.
- Eliminar el mueble i-ésimo de la cocina, si existe.
- Mover el mueble i-ésimo de la cocina (si existe) hacia la izquierda, hasta que se junte con el mueble (i - 1) ésimo o el extremo izquierdo de la pared.
- Destruir la cocina.

Especificación

Una cocina se define como una longitud positiva en la que se colocan una secuencia ordenada de muebles.

mueble: Un mueble es un elemento que puede colocarse en la cocina y que se define como una posición (distancia al extremo izquierdo de la cocina) y una anchura (longitud que ocupa a partir de la posición).

Operaciones:

Cocina(int tam);

Precondiciones: tam ≥ 0

Postcondiciones: Crea una cocina de longitud tam

bool Colocable(int anc, int p) const;

Precondiciones: La anchura es válida (no negativa, no mayor que la cocina, etc) y p es una posición válida de la cocina.

Postcondiciones: Devuelve true si el mueble se puede colocar en esa posición y false de lo contrario

void Colocar(int anc, int p);

Precondiciones: El mueble de posición p y anchura anc es Colocable

Postcondiciones: Añade un mueble de dichas características a la cocina.

mueble elemento(int i);

Precondiciones: $0 \leq i < nMuebles$

Postcondiciones: Devuelve el mueble i-ésimo

//Nota: Los muebles se contarán como 0, 1, 2, ...

void eliminar(int i);

Precondiciones: $0 \leq i < nMuebles$

Postcondiciones: elimina el mueble i-ésimo de la cocina. Si no existe dicho mueble no hace nada.

void desplazar (int i);

Precondiciones: $i \geq 0$ y el mueble i-esimo existe: $0 \leq i < nMuebles$

Postcondiciones: Entre el comienzo del mueble i-ésimo y el final del anterior no hay espacio

~Cocina();

Postcondiciones: Elimina la cocina y la memoria ocupada por sus muebles

Métodos utilizados de TADs conocidos:

```
template <typename T> Lista <T>:: Lista ();

template <typename T>
Lista<T>:: posicion Lista <T>:: primera ()const ;

template <typename T>
Lista<T>:: posicion Lista <T>:: fin ()const ;

template <typename T>
Lista<T>:: posicion Lista <T>::siguiente ()const ;

template <typename T>
Lista<T>:: posicion Lista <T>:: anterior ()const ;

template <typename T>
void Lista <T>:: insertar ( const Tl x, Lista<T>:: posicion );
```

```
template <typename T>
void Lista<T>::eliminar(Lista<T>::posicion);

template <typename T>
const T& Lista<T>::elemento(Lista<T>::posicion);

template <typename T>
bool Lista<T>::vacia() const;

template <typename T>
Lista<T>::~Lista();
```

Implementación :

```
struct mueble } //Es público, el usuario lo conoce
    int pos, anchura
    mueble (int p, int a) : pos (p), anchura (a) {}
};

class Cocina }
public

    Cocina (int tam);
    bool Colocable (int anc, int p ) const;

    mueble elemento (int i) const;
    void eliminar (int i);
    void desplazar (int i);
    ~Cocina();
```

```
private:
    Lista<mueble> L;
    int nMuebles, longitud;
};
```

/* Debido a que se necesita usar en ocasiones el método anterior(), se usará una lista doblemente enlazada. Como la lista va a estar ordenada según la distancia a la pared izquierda (pos), no nos interesa una lista circular, por lo que la implementación que se usará será la de la lista doblemente enlazada con cabecera.
*/

Implementación de los métodos:

```
Cocina::Cocina(int tam): nMuebles(0), longitud(tam){}

bool Cocina::Colocable(int anc, int p) const{
  typename Lista<mueble>::posicion pos = L.primera();
  bool valido;

  while (L.elemento(pos) < p && pos != L.fin()){
    pos = L.siguiente(pos)
  }
  //Pos es justo la posición del que supera p
  if (p+anc > L.elemento(pos)||p+anc > longitud ){
  valido = false // No cabe por la derecha (hay otro mueble o
  }else{                 // fin de la cocina)
```

```
    pos = L.anterior();
    // pos ahora es el anterior. Si la pos tanchura de ese
  // mueble > p => no colocable


    int dest = L.elemento(pos).pos + L.elemento(pos).anc;
    if(dest > p)}
      valido = false; // no cabe por la izquierda
    {else}
      valido = true;
    }
  }
  return valido;
}

void Cocina::Colocar(int anc, int p){
  if(L.vacia()){
    L.insertar(mueble(p,anc), L.primera());

  {else}
     typename Lista<mueble>::posicion pos = L.primera();

    while (L.elemento(pos).pos < p && pos != L.fin()){
      pos = L.siguiente(pos);
    }
```

```
    L.insertar(mueble(p,anc),pos);
{
 nMuebles++;
{
```


```
 mueble  elemento (int i)}
   assert( i>=0 && i<nMuebles);
   int j=0;
   typename Lista<mueble>::posicion pos = L.primera();

   while ( j<i )}
     pos = L.siguiente(pos);
     j++;
   {
   return L.elemento(pos);
{

 void Cocina:: Eliminar (int i)}
   assert ( i>=0 && i<nMuebles );
   typename Lista<mueble>::posicion
   int j=0;

   while( j<i )}
     pos = L.siguiente(pos);
     j++;
   }
   L. eliminar(pos);
   nMuebles --;
{
```

```
void Cocina::Desplazar(int i){
  assert(i>=0 && i<nMuebles);
  typename Lista<mueble>::posicion pos=L.primera();
  typename Lista<mueble>::posicion anterior;
  int j;

  while(j<i){
    pos=L.siguiente(pos);
    j++;
  }
  if(j==0){ // Quiere desplazar el primer elemento
    L.elemento(pos).pos=0;
  }else
    anterior=L.anterior(L.elemento(pos));
  int dist=L.elemento(anterior).pos+L.elemento(anterior).anc;
    L.elemento(pos).pos=dist;
  }
}

Cocina::~Cocina(){
  L.~Lista();
}
```

Preguntas de Teoría

1. Explicar la importancia de la creación y la utilización de un TAD.

La abstracción en programación es fundamental para generar modelos conceptuales o abstractos de los problemas a resolver.

Esta abstracción se puede lograr de dos maneras: La abstracción operacional (funciones) y la abstracción de datos (mediante los TADS).

Con los TADS podemos definir nuevos tipos de datos a partir de los que nos proporciona el lenguaje y también podemos definir operaciones con ellos. Con los TADS ampliamos el lenguaje.

2. Dado un vector circular de tamaño N "hecho" de colas, ¿Cuántos elementos puede almacenar?

Eso depende de la implementación de las colas. Si se usa una implementación estática o pseudostática con tamaño M, se podrán almacenar $N \cdot M$ elementos. Si la implementación de las colas es dinámica no hay un límite prefijado.

*Nota suponemos que en cada una de las N posiciones del vector circular está ocupado por una cola, es decir, que no se deja ninguna posición vacía del vector circular para delimitar un inicio y un fin.

3. ¿Por qué se pone el nodo cabecera en la representación del TAD lista mediante una estructura enlazada?

Para que la implementación en C++ mediante una estructura enlazada sea consecuente con la especificación del TAD Lista, la posición de cada nodo se debe representar con un puntero al nodo anterior.

Todos los nodos tienen un predecesor excepto el primero, por lo que se añade el nodo cabecera antes del primer nodo y de esta forma también se puede representar la posición del primer "nodo con valor" como un puntero al nodo cabecera.

1) ¿Qué sucede si después de la ejecución de una determinada operación no se cumplen las precondiciones de la misma?

No pasa nada. Las precondiciones deben cumplirse antes de la ejecución de la operación y las postcondiciones después de la ejecución.

2) ¿Qué condición tiene que cumplir una lista para que la búsqueda sea logarítmica?

La función búsqueda tal y como la hemos implementado en todas las representaciones es de orden lineal ($O(n)$) ya que se realiza una búsqueda secuencial.

No obstante, como en las representaciones estática y pseudoestática las posiciones son enteros sin signo, se podría implementar una búsqueda dicotómica en el TAD ListaOrdenada que sí sería logarítmica.

3) ¿Es necesario en el TAD ColaCircular el nodo cabecera?

No. El nodo cabecera se usaba en el TAD lista para representar la posición de un nodo como un puntero al nodo anterior. Sin embargo, en una Cola no nos hace falta representar las posiciones de los elementos ya que las operaciones se realizan en los extremos, por lo tanto no es necesario.

4) Estructura de datos Bicola e implementar las operaciones void pop_fin() y void push_frente(const T& e)

Estructura de datos empleada:

* Suponemos que usamos una plantilla: "template <typename T>" ...

```
//En la parte privada de la clase Bicola:
struct nodo } // Definición de la estructura nodo
  T elto;
  nodo* sig;
  nodo* ant;
  nodo(const T& x, nodo* a, nodo* s): elto(x), ant(a), sig(s){}
};
// En la parte privada también:
nodo* inicio;
nodo* fin;
```

Implementación de los métodos pedidos.

```
template <typename T>
void Bicola<T>::pop_fin() {
    // Assert (!vacia()); No es obligatorio comprobar la
    // precondición
    nodo* q = fin;
    fin = fin->ant
    delete fin;
}
```

```
template <typename T>
void Bicola <T>:: push_frente(const T& x) {
  if (inicio == 0) { // Bicola vacía
    inicio = fin = new nodo (x, 0, 0)
  } else {
    inicio = inicio -> ant = new nodo (x, 0, inicio);
```

1. Con la representación de colas mediante una estructura enlazada, con puntero al final y circular, ¿Cuántos elementos pueden almacenarse?

No hay un límite prefijado de elementos, por lo que desde el punto de vista teórico no hay límite. Obviamente al usar esa representación del TAD Cola en una máquina real existirán limitaciones físicas

3. ¿Qué condición deben cumplir los elementos de una lista para poder realizar búsquedas en orden cuadrático?

* Orden cuadrático: $O(n^2)$
* Orden lineal: $O(n)$

Ninguna. Las búsquedas de todas las representaciones del TAD lista son de orden lineal y por lo tanto, también pertenecen al orden cuadrático ya que $O(n) \subset O(n^2)$

Resumen TADS

Tema 4: Tipos Abstractos de Datos

Abstracción

Es la capacidad intelectual para comprender fenómenos complejos, prescindiendo de los detalles irrelevantes y resaltando los importantes.

Maneras de Abstraer una realidad:

Aplicando distintos puntos de vista: Aspectos irrelevantes en una situación pueden ser muy importantes en otra.

Abstrayendo con mayor o menor intensidad: Cuantos más detalles se ignoren, mayor será el nivel de abstracción

La abstracción en programación:

- Mediante ella generamos un modelo conceptual o abstracto del problema a resolver.
- Los lenguajes de programación proporcionan los medios para implementar los modelos abstractos de los problemas: abstracción operacional (funciones) y abstracción de datos (TADs, lo que se estudia en esta asignatura).

Abstracción operacional:

Consiste en definir funciones para crear nuevas operaciones a partir de las propias del lenguaje.

Estas funciones se pueden utilizar sin conocer cómo están implementadas. Nos importa qué hacen en vez de cómo lo hacen (Ocultación de la información).

Añade nueva utilidad a la que ya nos proporciona el lenguaje.

Abstracción de datos:

El lenguaje de programación proporciona funciones y tipos de datos. Hemos visto que podemos ampliar las funciones y veremos que también los tipos de datos.

La abstracción de datos se consigue mediante:

- La definición de una colección de datos con las mismas características y especificación de sus operaciones y propiedades: Tipo Abstracto de Datos (TAD)

- Utilización de los nuevos tipos y sus operaciones según la especificación. Lo que nos interesa es qué son y qué operaciones admiten, no cómo se almacenan ni cómo se procesan: Ocultación de Información.

Amplía de nuevo el lenguaje.

Principio de Ocultación de Información

Es una consecuencia de la abstracción.

Por un lado, mediante la abstracción operacional, el usuario de una función solo necesita conocer su especificación (qué hace), pero no los detalles de la implementación (cómo lo hace).

Por otro lado, en la abstracción de datos, la ocultación de información permite separar la interfaz de un TAD (características y comportamientos visibles desde el exterior) de los detalles internos de la implementación (invisibles desde el exterior).

La interfaz de un TAD es un "contrato" de servicios con el usuario, donde se indica qué se puede hacer con ese TAD.

La ocultación de información disminuye las interdependencias entre los componentes de un sistema (reduce el acoplamiento entre ellos).

Independencia de la Representación

La abstracción de datos se desarrolla en 2 niveles:

Nivel conceptual o de especificación: Definición del dominio (colección de datos, es decir, valores que pueden tomar las variables) y de la interfaz (operaciones y sus propiedades).

Nivel de representación o de implementación: Definición de la estructura de datos que soporta el TAD e implementación de las operaciones de forma que exhiban las propiedades y el comportamiento especificados.

La especificación del TAD es independiente de cualquier representación del mismo que se pueda diseñar, es decir, puede haber más de una implementación que cumpla una misma especificación

Como usuarios de un TAD solo nos interesa la especificación, por lo que su uso es independiente de la implementación subyacente.

La independencia de la representación implica:

- Un mismo TAD se puede implementar con distintas representaciones.
- Los requisitos del problema o aplicación determinarán la representación más adecuada a utilizar (pero todas se usarán de la misma forma).

Genericidad

Un tipo abstracto genérico es el que define una familia de tipos abstractos con un comportamiento común, pero que se diferencian en detalles que no afectan a dicho comportamiento (esos detalles suelen ser el tipo de dato con el que trabajan).

La abstracción de datos permite crear un TAD para resolver un problema concreto, aunque es conveniente, si es posible, generalizar la definición de este TAD, con vistas a que él mismo u otro de la familia puedan servir más adelante en circunstancias similares.

La generalización se emplea principalmente en los tipos de datos contenedores (almacenan un conjunto de datos del mismo tipo o con el mismo comportamiento. Ej: Pila, Lista ...).

Descomposición Modular (Modularidad)

La descomposición de un problema en subproblemas ayuda a tratar su complejidad y conduce a dividir el programa que lo resuelve en componentes llamados módulos.

Una buena descomposición del programa da lugar a módulos con características funcionales independientes. Esto aumenta la cohesión interna de los módulos (agrupar todas las funciones relacionadas con un tema en un módulo) y favorece su reutilización.

Módulo: Unidad de organización de un programa que engloba un conjunto de entidades (encapsulamiento), como datos, tipos, instrucciones o funciones y que controla lo que se puede ver y utilizar desde otros módulos.

Una función es un módulo ya que encapsula un conjunto de declaraciones e instrucciones para realizar un proceso.

Un conjunto de tipos, datos y funciones que forman parte de un programa se pueden encapsular dentro de un módulo que se almacena en un fichero.

Encapsulamiento de un TAD

Un TAD se puede encapsular en un tipo de módulo llamado clase propio de los lenguajes orientados a objetos (similar a las estructuras en C, pero pueden encapsular también funciones).

Una clase permite encapsular datos y operaciones y proporciona un mecanismo para ocultar en su interior los detalles de la implementación y mostrar las operaciones de uso externo a través de su interfaz.

Se habla de miembros de una clase → funciones (métodos)
→ datos (atributos)

Encapsulamiento y ocultación de la información: Muy relacionados en el contexto de los TAD, pero no se deben confundir.

- Encapsulamiento: Acción y efecto de agrupar dentro de un módulo (clase) los datos y operaciones que forman un TAD.

- Ocultación de información: Acción y efecto de mantener en secreto aquellos aspectos de un TAD que no se necesitan conocer para utilizarlo en un programa.

Un TAD consta de 2 partes: Especificación e implementación.

Especificación de un TAD

Contiene toda la información que necesita conocer el usuario del TAD.

1. Definición del dominio o conjunto de datos del TAD
2. Especificación de las operaciones:

   Especificación sintáctica: Indica cómo usar las operaciones, es decir, la forma de escribir cada una, con su nombre, tipo de operandos y resultado.

   Especificación semántica: Expresa qué hace y qué propiedades tiene cada una, es decir, su significado.

La especificación como contrato:

Un usuario puede utilizar el TAD bajo las condiciones de la especificación

El diseñador está obligado a implementarlo de modo que se cumpla el comportamiento de la especificación.

Realizar la especificación:

El dominio se puede definir enumerando sus valores (rango), mediante otros dominios conocidos...

La sintaxis de las operaciones la describimos mediante el prototipo de la función que realiza la operación

Para la semántica de cada operación usaremos precondiciones y postcondiciones con la siguiente interpretación:

"Si se cumplen las precondiciones, se puede realizar la operación y al finalizarla se garantiza que se cumplirán las postcondiciones"

Selección de las operaciones

Al menos seleccionaremos un conjunto mínimo con estas características:

- Se escogen las operaciones básicas o primitivas que nos permitan crear a partir de ellas otras mas complejas, dentro o fuera del TAD.

- El subconjunto de operaciones permitirá generar todos los valores del dominio.

- El conjunto de operaciones deberá ser completo, es decir, hará posible implementar cualquier algoritmo para procesar los valores del dominio utilizando exclusivamente dichas operaciones.

Clasificación de las operaciones de un TAD

Constructoras: Generan valores nuevos pertenecientes al dominio

Destructoras: Liberan las posiciones de memoria ocupadas por valores previamente creados.

Todos los TADs tienen los dos tipos anteriores.

Observadoras: Devuelven el estado o el contenido de un valor o de un componente del mismo.

Modificadoras: Cambian el estado o contenido de un valor o componente del mismo.

Implementación de un TAD

Se elige la representación de los datos y operaciones del TAD en términos de los tipos de datos y operaciones disponibles en el lenguaje de programación. Debemos:

- Diseñar una estructura de datos que represente los elementos del dominio de del TAD

- Implementación de los algoritmos de tratamiento de dicha estructura que realicen las operaciones del TAD tal como se ha especificado.

Ojo: No confundir Tipo Abstracto de Datos con Estructura de Datos.

El adjetivo abstracto indica que un TAD es un modelo conceptual descrito por su especificación, que solo existe en la mente del programador.

Una estructura de datos es la representación de esa abstracción en un lenguaje de programación.

Estructuras de Datos: Se crean por agrupación de tipos de datos simples mediante los tipos estructurados del lenguaje

- Vectores y Matrices Conjunto de elementos del mismo tipo
- Registros o Estructuras Conjunto de elementos que pueden ser de distintos tipos
- Estructuras enlazadas mediante punteros: Sus componentes (nodos) son registros que incluyen punteros a las posiciones de memoria de otros nodos de la estructura.

Se pueden combinar estas estructuras para crear otras aún más complejas.

Clasificación de las Estructuras de datos:

- Estática: Tamaño fijo (establecido en tiempo de compilación) y se almacena en posiciones contiguas de memoria. Formadas por vectores/matrices y registros.
- Pseudoestática: Estructura estática cuyo tamaño se fija en tiempo de ejecución.
- Dinámica: Tiene un número variable de elementos almacenados en posiciones no contiguas de memoria y enlazados mediante punteros.

La elección de una u otra representación de un TAD se realiza considerando los requisitos de la aplicación y siguiendo los criterios de eficiencia en tiempo y espacio.

Técnica de diseño descendente: Consiste en ir descomponiendo los problemas haciéndolos cada vez más simples.

Programación Basada en TAD

La resolución de un problema mediante un programa la enfocaremos como un proceso de abstracción y descomposición del problema que lleva a la construcción de un modelo formal basado en tipos abstractos de datos.

El concepto de TAD proporciona una base ideal para la descomposición modular de un programa grande.

Una vez identificados los TAD que forman parte de la solución, se crean sus especificaciones y el programa se escribe a partir de ellas.

Por último, se elige una representación para cada TAD y se implementan las operaciones mediante los algoritmos más eficientes según la estructura de datos elegida.

Conclusiones

Con una metodología de programación basada en tipos abstractos de datos se logra disminuir la complejidad inherente a la construcción de programas, separando dos tareas independientes:

1. Construir un programa a partir de unos objetos adecuados a las características del problema a resolver.
2. Implementar estos objetos a partir de los elementos del lenguaje.

Especificación de un TAD

1.- Descripción. Debe quedar definido el dominio

2.- Listado de métodos (incluido constructores y destructor) siguiendo la siguiente estructura

- Cabecera en el lenguaje de programación (C++)
- Precondiciones: Nos fijamos en los parámetros de entrada (incluido el objeto desde el que se llame al método).
- Postcondiciones: Nos fijamos en los parámetros de salida (incluidos los cambios que se realicen en en el objeto desde el que se hace la llamada).

Métodos observadores: Cuando no realizan ningún cambio en el objeto desde el que se hace la llamada, se les añade "const". Ej: long::racional num() const;

Paso por referencia constante: Se pasa por referencia (más eficiente) pero se prohibe modificar el valor. Ej: const int& a

Destructor: En C++ todas las clases tienen un destructor por defecto. Este libera la memoria que ocupan los tipos que forman el TAD. Sin embargo, muchas veces en la instancia del TAD solo tendremos un puntero a una estructura mayor, pero el destructor por defecto solo borra el puntero, no la memoria a la que apunta.

Operaciones primitivas: Cada una se encarga de una sola cosa. Son lo más simple posible.

Constructores: En C++, tienen una lista de inicialización, separada por dos puntos ":" entre los paréntesis de los parámetros y la llave con la que comienza el cuerpo del método.

Tema 5 TAD Pila

Definición

Secuencia de elementos de un tipo determinado en la que todas las operaciones se realizan en un extremo de la misma, llamado tope o cima. El último elemento añadido es el primero en salir (LIFO).

Implementación estática: Vector y un entero que almacena qué elemento es el tope

Plantillas en C++

Una plantilla es una definición genérica de una familia de clases (o funciones) que difieren en detalles (como los tipos de datos usados) de los cuales no depende el concepto representado. A partir de la plantilla el compilador puede generar una clase o función específica.

Con ellas realizaremos una implementación genérica del TAD (pila) y en los programas usaremos las clases que genera el compilador automáticamente (pila de int, char...).

Al definir una plantilla se presuponen ciertas propiedades como que el tipo de dato tenga un constructor por defecto o que se pueda comparar con los operadores relacionales (>, <, ...). De lo contrario hay errores de compilación.

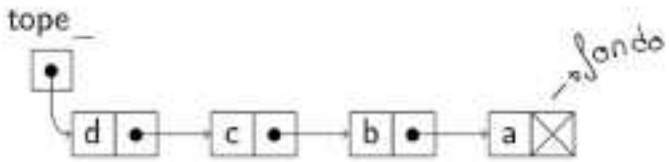
Aunque normalmente el código se divide en 2 ficheros: .h (cabeceras) y .cpp (implementación), al usar plantillas metemos todo en el .h ya que si no el compilador hace cosas raras.

Las plantillas sirven para hacer programación genérica: Principio de genericidad.

Implementación Pseudoestática: Al constructor se le pasa como parámetro la capacidad máxima.

Implementación dinámica (celdas enlazadas):

Estructura enlazada (para hacer una pila dinámica)



La pila queda representada por un puntero (tope) = por el que comienza.

Debemos definir una estructura nodo que tenga un campo del tipo de elemento y otro un puntero a nodo. También debe tener un constructor.

En C++ struct y class se pueden usar para lo mismo. En struct los elementos por defecto son públicos y en class privados.

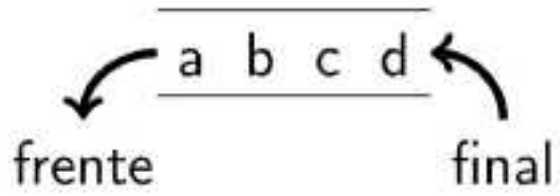
Tema 6: TAD Cola

Definición

Secuencia de elementos de un tipo determinado en la que las operaciones se realizan por los extremos.

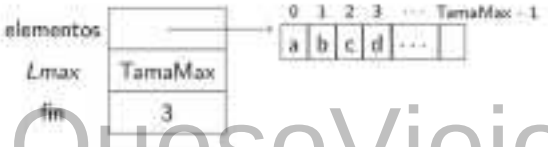
- Las eliminaciones se realizan por el extremo llamado inicio, frente o principio de la cola.
- Los nuevos elementos son añadidos por el otro extremo, llamado fondo o fin de la cola.

El primer elemento añadido es el primero en salir (FIFO)

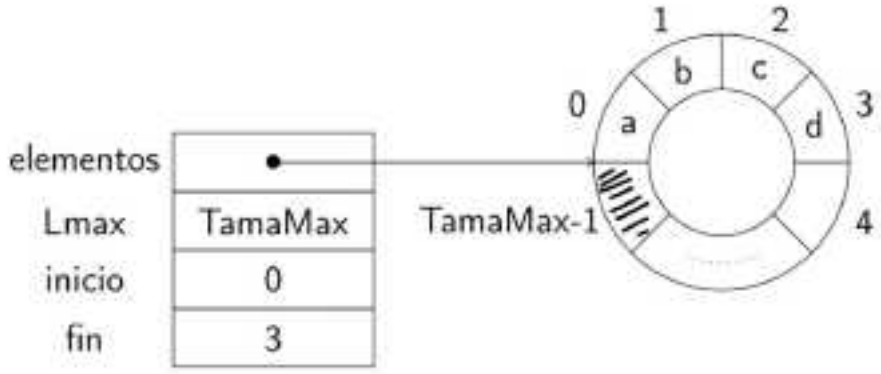


Implementación Pseudostática: Un vector de tamaño n, un entero LMax con valor n y un entero tope que contiene el índice con el que acceder al elemento fondo en el vector.

Operaciones: Pop y push: Una de O(1) y otra O(n)



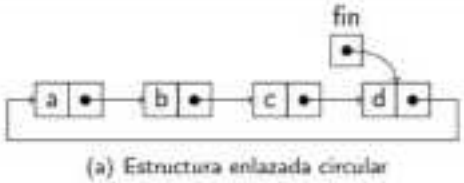
Implementación vectorial circular: Vector, Lmax y 2 enteros que indican el tope y el fondo. Push y Pop ∈O(1)



Hay que dejar una posición sin elemento, de lo contrario no podremos distinguir cuándo está llena y cuándo está vacía.

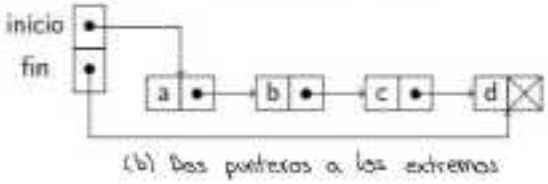
Implementación dinámica:

Opción 1:



(a) Estructura enlazada circular

Opción 2:



(b) Dos punteros a los extremos

Bicola: Una posible implementación sería como la opción 2 anterior, con 2 punteras en cada nodo, uno al siguiente y otro al anterior, para que hacer pop al final o push al principio sea O(1)

Tema 7: TAD Lista

Definición

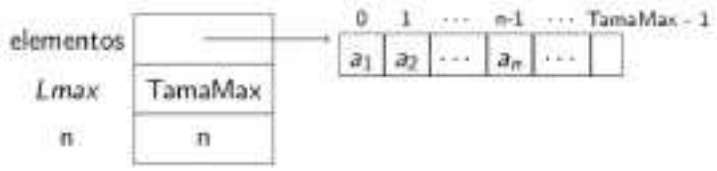
Secuencia de elementos de un tipo determinado cuya longitud $n \geq 0$ es el número de elementos que contiene. Si $n=0$ (no hay elementos), se denomina lista vacía.

$$L = (a_1, a_2, \ldots, a_n)$$

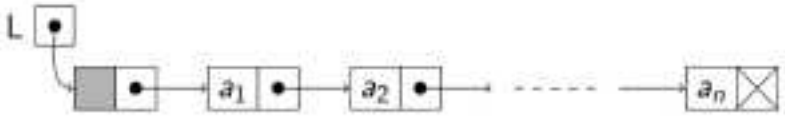
Posición: Lugar que ocupa un elemento en la lista. Los elementos están ordenados de manera lineal según las posiciones que ocupan. Todos los elementos salvo el primero tienen un único predecesor y todos excepto el último tienen un único sucesor.

Posición fin(): Posición especial que sigue a la del último elemento y que nunca está ocupada por elemento alguno.

Implementación pseudoestática Vector de elementos, tamaño máximo y longitud actual. Insertar y eliminar $\in O(n)$



Implementación dinámica (estructura enlazada con cabecera): El tamaño de la estructora de datos (Lleva en la otra versión) varía en tiempo de ejecución con el tamaño de la lista (n). Hay que reservar espacio en cada nodo para los enlaces.



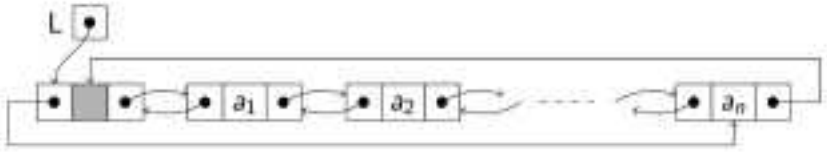
Posición de un elemento: Puntero al nodo anterior.

Primera posición: Puntero al nodo cabecera.

Posición fin(): Puntero al último nodo de la estructura.

Como la posición es un puntero al anterior, hay que meter un nodo antes del primero (ese nodo no tiene valor y se llama "nodo cabecera").

Las operaciones buscar(), anterior() y fin() son O(n), el resto O(1)
Si hiciera una estructura circular (fin->sig = cabecera) y L apuntara a fin(), entonces fin() ∈ O(1)
Implementación doblemente enlazada con cabecera:



Posición de un elemento: Puntero al nodo anterior.

Primera posición: Puntero al nodo cabecera.

Posición fin(): Puntero al último nodo de la estructura.

Simplemente ahora tenemos 2 punteros, anterior() ∈ O(1)

Implementación circular: Como la anterior pero sin cabecera.

Notas sueltas:

* Constructor de copia:

Para llamar al constructor de copia de un tad llamado tad_Ej, se hace en la declaración de una nueva instancia de ese tad y ya debía existir la que va a ser copiada.

```
tad_Ej p; ——> Si estoy llamando al constructor vacío,
                no pongo paréntesis
tad_Ej q(p);
```

Es obligatorio crearlo ya que al pasar o devolver un parámetro por valor se llama a este método.

* Destructor: Es un método que se llama ~NombredelTAD. En mi ejemplo: ~tad_Ej

Para tipos primitivos como **int**, se libera la memoria sin más. En un TAD, si tenemos campos que sean vectores o punteros, hay que crear un método destructor que libere esa memoria con los métodos **delete[]** nombre_vector y **delete** nombre_puntero. El puntero en sí se elimina, pero no la memoria a la que apunta. Usando los delete liberamos la memoria apuntada por los punteros. La variable puntero no se ha eliminado todavía, por lo que puedo reutilizarla.

* Si se apunta a un objeto, se llama previamente al destructor.

Si el puntero es un puntero nulo, delete no hace nada.

Antes de que se cierre el ámbito de esa instancia del TAD y que se deba liberar la memoria, se llama al destructor así:

```
p.~tad_Ej();
```

* Clase string # include <string>

- **Declaración de una cadena:** `string cad1;`

  No se especifica tamaño

- **Lectura de una cadena por teclado:**

  Usamos la función getline que está sobrecargada para las cadenas: `getline(cin, cad1);` (flujo de entrada)

  Esta función no deja '\n' al final

  getline is non-member function ⇒ no se llama a partir de una instancia (no se pone cad1.getline)

- **Tamaño de la cadena:**

  Llamamos a la member function length: `cad1.length();`

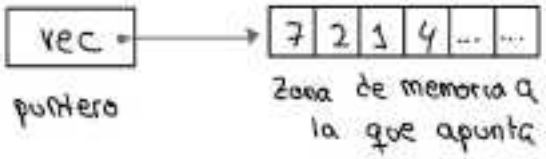
  size() y length() hacen lo mismo.

  Devuelven el nº de Bytes. Los caracteres ASCII ocupan 1B así que en principio no hay problema.

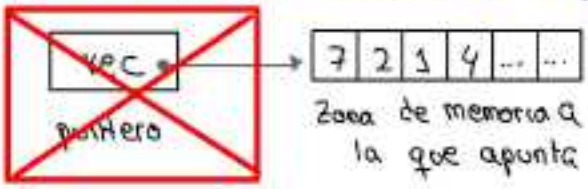
  cad1 = "Hola" ⇒ cad1.size() devuelve 4

* Estructuras de memoria dinámica

Los punteros ni se asignan ni se destruyen bien por defecto, ya que por defecto no se trabaja con la zona de memoria a la que apuntan. Luego el operador de asignación, el constructor de copia y el destructor son métodos que debemos implementar.

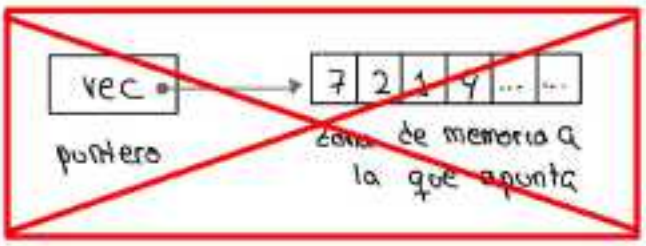


Comportamiento del destructor por defecto:



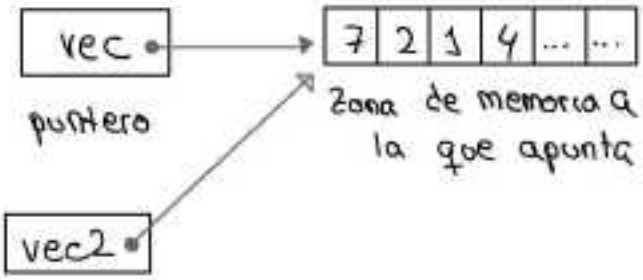
Elimina solo la variable puntero

Comportamiento deseado:



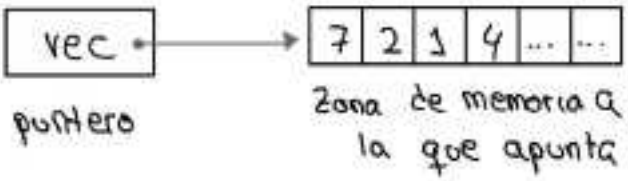
Debe liberar la memoria a la que apunta y luego eliminar la variable puntero.

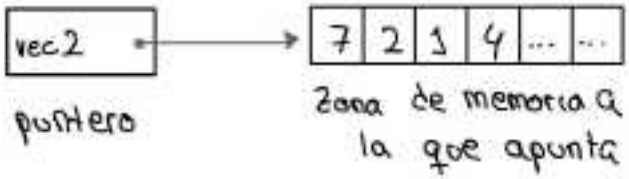
Comportamiento del constructor de copia por defecto



Si liberamos la memoria de uno, como es la misma, la liberamos para ambos

Comportamiento deseado:

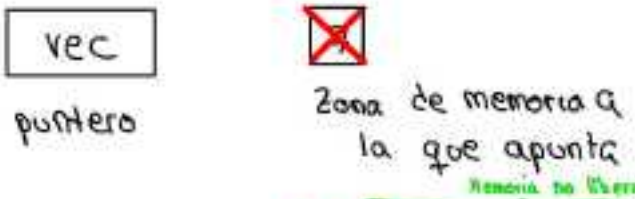




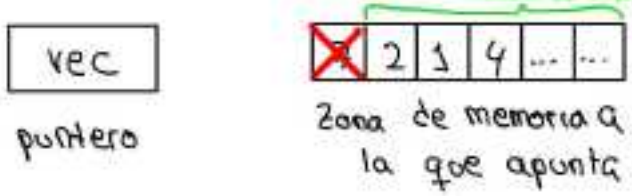
Se hace una copia de la memoria y la variable puntero que se crea apunta a la copia. Si eliminamos la memoria para uno, el otro sigue existiendo.

Comportamiento de los métodos delete:

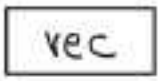
delete vec



delete vec



delete [] vec



puntero

Zona de memoria a la que apunta

* Si vec es un vector de objetos, primero se llama al destructor con cada uno.

Con estos métodos solo se libera la memoria. Ahora puedo hacer que el puntero apunte a otro sitio.

Si antes de que el puntero apunte a otro sitio no libero la memoria, esa memoria quedaría ocupada y sin direccionar y eso no es bueno.

- Reservar memoria dinámica:

  new tipo devuelve la dirección de memoria de un bloque de memoria requerido para el tipo.

  Si se trata de un tipo del lenguaje, se puede escribir un valor de inicialización entre paréntesis:

  new tipo (inicializador)

  Si el tipo es una clase, la memoria reservada se inicializa con el constructor de la clase. Si hay que pasarle una lista de parámetros, se pasan entre paréntesis.

  new clase(param1, param2, ...)

`new tipo[n]` Reserva memoria dinámica para un vector de n elementos de tipo y devuelve su dirección.

Si el tipo es una clase, cada posición se inicializa con el constructor predeterminado.

Paso por parámetros de un objeto (Pila, Cola...)

Por defecto se pasa por valor. Para pasar por referencia:

Llamada: funcion(C)

Prototipo: tipo funcion(Cola<T> & C)

Fichero.h: Al usar nuestro TAD en varios módulos (hacer #include "Ficheros.h"), el compilador lo incluiría varias veces, por lo que tendríamos varias veces definida esa clase. Para evitarlo, todo el .h se engloba en:

```
#ifndef nombre
#define nombre
   : //código del .h
#endif
```

Donde "nombre" sería un flag que le indica al compilador que ya ha sido definida esa clase.

TADs

- Pila
  - pseudo
  - dinamica
- Cola
  - Pseudo
  - Pseudo-Circular
  - Dinámica
    - 2 punteros
    - Circular y 1 puntero
- Bicola
- Lista
  - Pseudoestáticas
  - Dinámica
    - Cabecera
      - x2 enlazada
      - Cabecera-circular
  - Circular
  - Ordenada