TAD PILA

Una **pila** es una secuencia de elementos en la que todas las operaciones se realizan por un extremo de la misma. Dicho extremo recibe el nombre de **tope.**

En una pila el último elemento añadido es el primero en salir de ella, por lo que también se les conoce como estructuras **LIFO** *(Last Input First Output).*

Definición:

Una pila es una secuencia de elementos de un tipo determinado, en la cual se puede añadir y eliminar elementos sólo por uno de sus extremos llamado *tope* o *cima.*

Operaciones:

Pila()

Post: Crea una pila vacía.

bool vacia() const

Post: Devuelve **true** si la pila está vacía.

const tElemento& tope() const

Pre: La pila no está vacía.

Post: Devuelve el elemento del tope de la pila.

void pop()

Pre: La pila no está vacía.

Post: Elimina el elto. del tope de la pila y el sig. pasa a ser el nuevo tope.

void push(const tElemento& x)

Post: Inserta el elto. *x* en el tope de la pila y el antiguo tope pasa a ser el sig.

Implementación:

- Vectorial estática:

```
const int LMAX = 100;       // Longitud máxima de una pila.
private:
      tElemento elementos[ LMAX ];     // Vector de elementos.
      int tope_;                       // Posición del tope.
```

- Vectorial pseudoestática:

```
private:
      tElemento *elementos;       // Vector de elementos.
      int Lmax;                   // Tamaño del vector.
      int tope_;                  // Posición del tope.
```

- Mediante celdas enlazadas:

```
private:
      struct nodo {
            T elto;
            nodo *sig;
            nodo(const T& e, nodo *p = 0): elto(e), sig(p) {}
      };

      nodo *tope_;
```

TAD COLA

Una **cola** es una secuencia de elementos 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 al final por el otro extremo, llamado **fondo** o final de la cola.

En una cola el primer elemento añadido es el primer elemento en salir de ella, por lo que también se les conoce como estructuras **FIFO** (*First Input First Output*).

Definición:

Una cola es una secuencia de elementos de un tipo determinado, en la cual se pueden añadir elementos sólo por un extremo, al que llamaremos fin, y eliminar por otro, al que llamaremos inicio.

Operaciones:

Cola()

Post: Crea una cola vacía.

bool vacia() const

Post: Devuelve **true** si la cola está vacía.

const tElemento& frente() const

Pre: La cola no está vacía.

Post: Devuelve el elemento del inicio de la cola.

void pop()

Pre: La cola no está vacía.

Post: Elimina el elemento del inicio de la cola y el siguiente se convierte en el nuevo inicio.

void push(const tElemento& x)

Post: Inserta el elemento *x* al final de la cola.

Implementación:

- Vectorial pseudoestática:

```
private:
      tElemento *elementos;      // Vector de elementos.
      int Lmax;                  // Tamaño del vector.
      int fin;                   // Posición del último.
```

- Vectorial circular:

```
private:
      tElemento *elementos;      // Vector de elementos.
      int Lmax;                  // Tamaño del vector.
      int inicio, fin;      // Posiciones de los extremos
                                 // de la cola.
```

  Nota: Conservamos una posición entre el inicio y el fin para saber si la cola se encuentra vacía/llena.

```
bool vacia() const{return ((fin + 1) % Lmax == inicio);}
bool llena() const{return ((fin + 2) % Lmax == inicio);}
```

- Estructura dinámica: El tamaño de la estructura varía en tiempo de ejecución con el tamaño de la cola. A cambio se ocupa espacio adicional con los enlaces. (Enlazada circular, o con dos punteros a los extremos).

```
private:
      struct nodo {
            T elto;
            nodo *sig;
            nodo(const T& e, nodo *p = 0): elto(e), sig(p) {}
      };

      nodo *inicio, *fin;        // Dos punteros a los
                                 // extremos de la cola.
```

TAD LISTA

Definición:

Una **lista** es una secuencia de elementos de un tipo determinado $L = (a_1,\ a_2,\ ...,\ a_n)$ cuya **longitud** es n $\geq$ 0. Si n = 0, entonces es una **lista vacía.**

Posición

Lugar que ocupa un elemento en la lista. Los elementos están ordenados de forma 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 ningún elemento.

Operaciones:

Lista()

Post: Crea y devuelve una lista vacía.

void insertar(const T& x, posicion p)

Pre: $L = (a_1,\ a_2,\ ...,\ a_n)$

$1 \leq p \leq n+1$

Post: $L = (a_1, ..., a_{p-1}, x, a_p, ..., a_n)$

void eliminar(posicion p)

Pre: $L = (a_1,\ a_2,\ ...,\ a_n)$

$1 \leq p \leq n$

Post: $L = (a_1, ..., a_{p-1}, a_{p+1}, ..., a_n)$

const T& elemento(posicion p) const

T& elemento(posicion p)

Pre: $L = (a_1,\ a_2,\ ...,\ a_n)$

$1 \leq p \leq n$

Post: Devuelve a_p, el elemento que ocupa la posición *p* de la lista *L.*

posicion buscar(const T& x) const

Post: Devuelve la posición de la primera ocurrencia de *x* en la lista. Si *x* no se encuentra, devuelve la posición fin().

posicion siguiente(posicion p) const

Pre: $L = (a_1, a_2, ..., a_n)$
$1 \leq p \leq n$

Post: Devuelve la posición que sigue a *p.*

posicion anterior(posicion p) const

Pre: $L = (a_1, a_2, ..., a_n)$
$2 \leq p \leq n + 1$

Post: Devuelve la posición que precede a *p.*

posicion primera() const

Post: Devuelve la primera posición de la lista. Si la lista está vacía, devuelve la posición *fin().*

posicion fin() const

Post: Devuelve la última posición de la lista, la siguiente a la del último elemento. Esta posición siempre está vacía, no existe ningún elemento que la ocupe.

Implementación:

- Vectorial pseudoestática:

```
private:
      T *elementos;    // Vector de elementos.
      int Lmax;        // Tamaño del vector.
      int n;           // Longitud de la lista.
```

- Estructura enlazada (dinámica): El tamaño de la estructura de datos varía en tiempo de ejecución con el tamaño de la lista. A cambio se ocupa espacio adicional con los enlaces.

  Representación de posiciones:

  - *Posición de un elemento.* Puntero al nodo que lo contiene.
  - *Primera posición.* Puntero al primer nodo de la estructura.
  - *Última posición (fin()).* Puntero almacenado en el último nodo de la estructura, o sea, un puntero nulo.

```
typedef nodo *posicion;     // Posición de un elto.
private:
      struct nodo{
            T elto;
            nodo *sig;
            nodo(T e, nodo *p = 0): elto(e), sig(p){}
      };

      nodo *L;         // Lista enlazada de nodos.
```

  Inserción y eliminación de elementos.
  1- El parámetro posición de estas operaciones se pasa por referencia, porque un nuevo elemento ocupará dicha posición al finalizar.
  2- No se cumple totalmente con la especificación del TAD, porque este parámetro se pasa por valor.
  3- Por ello, el uso con esta implementación provocará errores que no se producirán con otras implementaciones.

  No se pueden pasar por referencia a insertar() y eliminar() las posiciones devueltas por fin(), primera(), anterior(), siguiente() o buscar().

- Estructura enlazada con cabecera: Para solventar el incumplimiento de la especificación y la ineficiencia de las inserciones y eliminaciones cambiamos el modo de representar las posiciones.

  Representación de posiciones:
  - *Posición de un elemento.* Puntero al nodo anterior.
  - *Primera posición.* Puntero al nodo cabecera.
  - *Última posición (fin()).* Puntero al último nodo de la estructura.

```
typedef nodo *posicion;     // Posición de un elto.
private:
      struct nodo{
            T elto;
            nodo *sig;
            nodo(const T& e, nodo *p = 0): elto(e), sig(p){}
      };

      nodo *L;         // Lista enlazada de nodos.
```

  Eficiencia:
  - Las operaciones *buscar(), anterior() y fin()* son $\Theta(n)$.
  - El resto de operaciones son $\Theta(1)$.

- Estructura enlazada circular: La operación *fin()* se puede hacer de $\Theta(1)$ sin alterar la eficiencia de las otras operaciones y sin usar espacio adicional, debido al acceso directo al último nodo.

- Estructura doblemente enlazada con cabecera:
  Representación de posiciones:
  - *Posición de un elemento.* Puntero al nodo anterior.
  - *Primera posición.* Puntero al nodo cabecera.
  - *Última posición (fin()).* Puntero al último nodo de la estructura.

```
typedef nodo *posicion;     // Posición de un elto.
private:
      struct nodo{
            T elto;
            nodo *ant, *sig;
            nodo(const T& e, nodo *a = 0, nodo *s = 0) :
            elto(e), ant(a), sig(s){}
      };
      nodo *L;         // Lista doblemente enlazada de nodos.
```

TAD LISTA CIRCULAR

Definición:

Una **lista circular** es una secuencia de elementos de un mismo tipo en la que todos tienen un predecesor y sucesor, es decir, es una secuencia sin extremos. Su **longitud** coincide con el número de elementos que la forman; si es 0, entonces la lista está vacía. Una lista circular de longitud *n* se puede representar de la forma $L = (a_1, a_2, \ ..., a_{n,} a_1)$ donde repetimos a_1 después de a_n para indicar que el elemento que sigue a a_n es a_1 y el anterior a éste es a_n.

Definimos una **posición** como el lugar que ocupa un elemento en la lista. La constante **POS_NULA** denota una posición inexistente.

Al no tener extremos, no existen posiciones inicial y final. En consecuencia, las operaciones del TAD serán las mismas a las del TAD Lista a excepción de *primera()* y *fin()*.

Para recorrer esta lista, necesitaremos de una posición desde la cual comenzar el recorrido. Esta **posición** nos la de la operación que llamaremos *inipos().*

Operaciones:

ListaCir()

Post: Crea y devuelve una lista circular vacía.

void insertar(const T& x, posicion p)

Pre: $L = ()$ y *p* es irrelevante, o bien $L = (a_1, a_2, \ ..., \ a_n, a_1)$ y $1 \le p \le n$.

Post: Si $L = ()$, entonces $L = (x, x)$ (Lista circular con un único elemento); en caso contrario, $L = (a_1, \ ..., \ a_{p-1}, x, a_p, ..., a_n, a_1)$.

void eliminar(posicion p)

Pre: $L = (a_1, a_2, \ ..., \ a_n, a_1)$; $1 \le p \le n$.

Post: $L = (a_1, \ ..., \ a_{p-1}, a_{p+1}, ..., a_n, a_1)$.

const T& elemento(posicion p) const

T& elemento(posicion p)

Pre: $L = (a_1, a_2, \ ..., \ a_n, a_1)$; $1 \le p \le n$.

Post: Devuelve a_p, el elemento que ocupa la posición *p.*

posicion buscar(const T& x) const

Post: Devuelve la posición de una ocurrencia de *x* en la lista. Si *x* no pertenece a la lista, devuelve *POS_NULA.*

posicion inipos() const

Post: Devuelve una posición indeterminada de la lista. Si la lista está vacía, devuelve *POS_NULA*. Esta operación se utilizará para inicializar una variable de tipo **posicion**.

posicion siguiente(posicion p) const

Pre: $L = (a_1, a_2, \ ..., \ a_n, a_1)$; $1 \leq p \leq n$.

Post: Devuelve la posición siguiente a p.

posicion anterior(posicion p) const

Pre: $L = (a_1, a_2, \ ..., \ a_n, a_1)$; $1 \leq p \leq n$.

Post: Devuelve la posición anterior a *p.*

Implementación:

- Estructura doblemente enlazada. No se necesita de nodo cabecera, ya que cualquier nodo de la estructura está seguido (y por tanto precedido) de otro.
  Representación de posiciones:
  - *Posición de un elemento.* Puntero al nodo anterior.
  - *inipos().* Puntero a algún nodo de la estructura, por ejemplo, **L**.