MODULO 3

OBJETIVOS

Al terminar este módulo el estudiante deberá:

- Comprender la especificación de los tipos abstractos lista, lista ordenada, stack y colas y reconocer su utilidad.
- Emplear adecuadamente las operaciones sobre listas dadas en la especificación para codificar programas de manejo de listas.

CONTENIDO

1 El tipo de dato lista
1.1 General
1.2 Ordenada
1.3 Stack
1.4 Queue

2 Especificaciones
2.1 Especificación general de la lista
2.1 Especificación stack (pilas)
2.2 Especificación queue (colas)

3 Programacion con listas en C++

A Referencias

B Taller

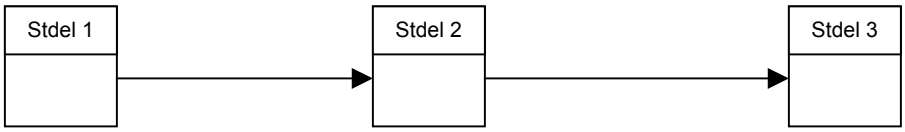
C Laboratorio

1 EL TIPO DE DATO LISTA

1.1 LISTA GENERAL

La lista es un conjunto de componentes estándar con estructura lineal (los enlaces entre los elementos son de uno a uno, cada elemento tiene un y sólo un predecesor un y sólo un sucesor, excepto el primer elemento que no tiene predecesor y el último que no tiene sucesor) Una lista contiene n elementos, donde n varía a través del tiempo a medida que se insertan y borran elementos. Se dice que los elementos almacenados en la lista son de tipo *stdelement* (elemento estándar, como se vio antes, *stdelement* contiene un *campo clave* que lo identifica de manera única.

Figura 1.1 Lista



1.2 LISTA ORDENADA

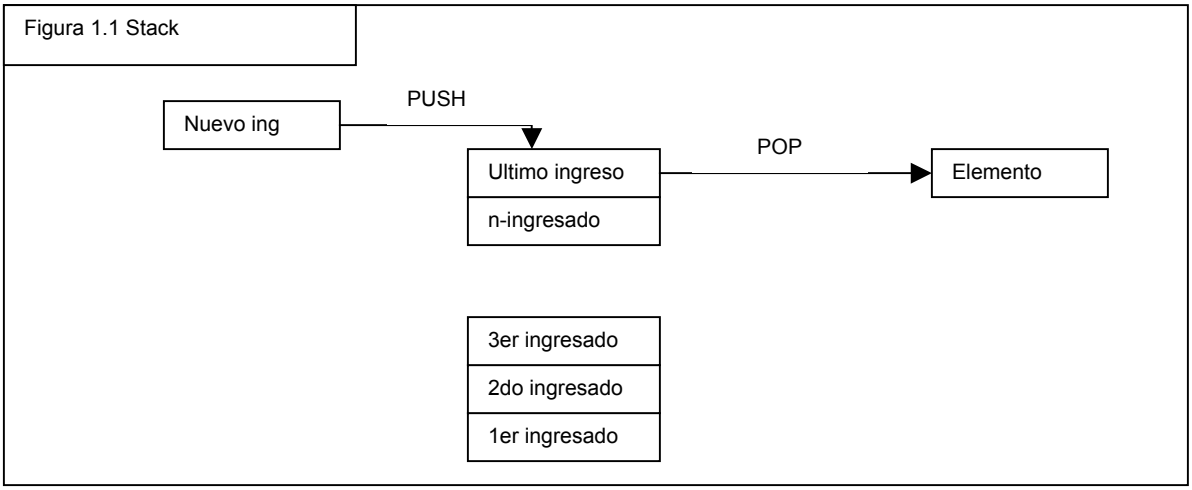
La lista ordenada es un tipo de lista que se diferencia de la lista general sólo en que, como su nombre lo dice, están ordenados y dicha ordenación está definida como parte de la especificación. Para asegurar que se mantiene el correcto orden de los elementos el control del punto de inserción es tomado del usuario y colocado en la implementación, lo cual garantiza la integridad de la lista *ordenada*.

Inicialmente, se pueden considerar tres tipos de listas ordenadas:

- Listas ordenadas cronológicamente.
- Listas ordenadas por campo clave.
- Listas ordenadas por la frecuencia.

1.3 STACKS (PILAS)

Los stacks (pilas) son tipos de datos que tienen un comportamiento LIFO (last in, first out) que significa: el último elemento en entrar es el primero en salir. Se pueden añadir elementos a la pila, sin embargo, el único elemento que puede ser removido, es el que ha ingresado más recientemente (*last in)* Cuando el último elemento ingresado (*last in)* es removido, el elemento que fue ingresado antes que este pasa a ser el ingresado más recientemente (*last in*).

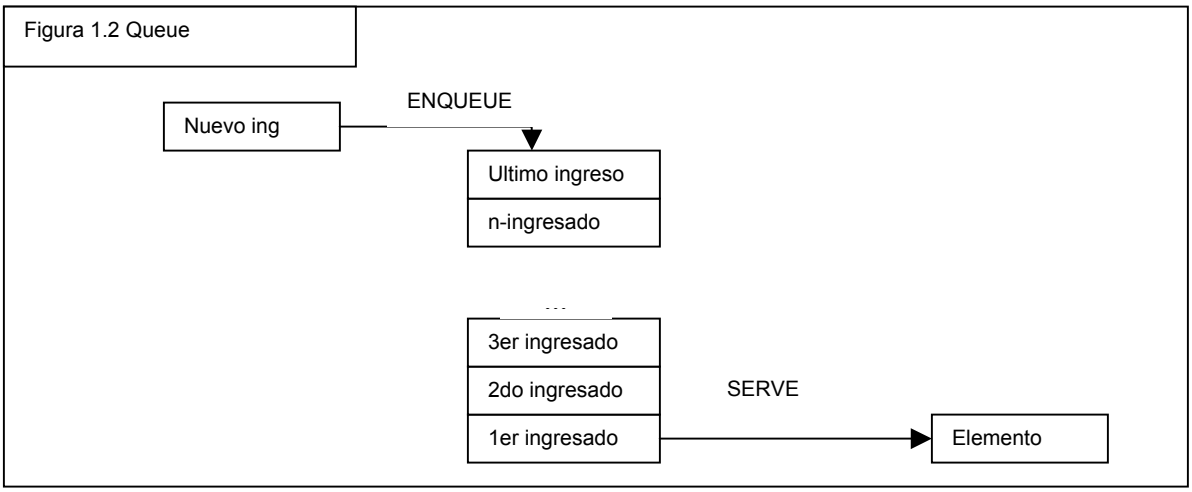


La operación para ingresar elementos al stack es comúnmente llamada *push.* Y la función para retirar el elemento más recientemente ingresado (last in) es usualmente llamada *pop.*

1.4 QUEUES (COLAS)

La cola es un tipo de dato que exhibe comportamiento FIFO (first in, first out). Los elementos pueden ser adicionados a la cola, sin embargo el único elemento que puede ser retirado es el menos recientemente ingresado, en otras palabras el que primero ha arribado a la cola. Cuando se retira el que ha arribado de primero, el siguiente arribó más próximo se convierte en el elemento adicionado menos recientemente.

La operación de agregar un elemento a la cola es comúnmente llamado *enqueue,* la operación de retirar el elemento menos recientemente ingresado, es usualmente llamada *serve.*



2 ESPECIFICACIÓN DEL TIPO DE DATO LISTA

2.1 ESPECIFICACIÓN DE LA LISTA GENERAL

A continuación se presenta la especificación del tipo de dato abstracto lista, de acuerdo a la definición dada en la sección anterior. En esta ocasión se usa lógica formal en la especificación.

Especificación TD lista:

Caracterización: Agregado iterado con componentes de tipo estándar.

Componentes: Conjunto de elementos estándar (clave + "struct")

Estructura:

- Una lista tiene un conjunto E de n elementos estándar ($n \in \{0,1,2,....\}$).
- Hay una relación dirigida entre los elementos, denominada "sucesor" (i.e. hay una función no reflexiva, definida sobre todos menos un elemento, y que relaciona un elemento con otro diferente)
- Si n>0, hay un elemento que no es sucesor de ningún otro y lo denominamos "primero".
- Si n>0 hay un elemento que no tiene sucesor y lo denominamos "último".
- Todos los demás tienen un sucesor diferente a ellos, y son el sucesor de otro diferente a ellos.
- Hay una correspondencia de los elementos de E con los n primeros números cardinales, y al número que le corresponde a un elemento lo denominaremos su "posición en la lista" (todo elemento tiene posición, pero solo hay elemento para las posiciones {1,..,n})

Definiciones:

Primero: Elemento de la lista que no es sucesor de ningún otro.
Ultimo: Elemento de la lista que no tiene sucesor.
Siguiente de e: Elemento sucesor de e.
Anterior a e: Elemento del que e es el sucesor.
Adelante: Relación dirigida entre $e \in E$ y $e' \in E$ y $e' \neq e$:
1-: e'=siguiente(e) $\Rightarrow$ e'=adelante(e)
2-: $(\exists e'' \in E$ / (e''=adelante(e)$\wedge$e'=siguiente(e'')) $\Rightarrow$ e'=adelante(e)
Atrás: Relación dirigida entre $e \in E$ y $e' \in E$ y $e' \neq e$:
1-: e'=adelante(e) $\Rightarrow$ e=atras(e')
Posición de e: |{e': $e' \in E$ / e'=atrás(e)}|+1.
Camino a e: Lista de los antecesores a e bajo la relación "sucesor".
actual: El elemento señalado en la lista.
Cursor: Posición del elemento señalado en la lista.
Lista vacía: Lista con n=0.

Literal:

- {<lit(e1)>,<lit(e2)>,...<lit(en)>} para lista no vacía

- {NULL}.para lista vacía.

Operaciones:

Definición de variables:

- L: una lista.
- Cursor: entero, posición del elemento señalado.
- POF: una lista.
- PRF: una lista.
- e: un elemento del tipo del componente.
- a: un elemento del tipo del componente.
- n: un entero (el tamaño de la lista) n=tamaño(L).
- k: una clave del tipo de la clave del componente.
- t: un entero.
- i: un entero.

Creadora y Destructora:

- **L=L.nueva_lista()**: Crea la lista L como vacía.
  - PRE: $\neg\exists L$
  - POST: $\exists L \wedge L=\{NULL\} \wedge Cursor=0$
- **L.destruir_lista**(*L): Destruye la lista L.
  - PRE: L={NULL}
  - POST: $\neg\exists L$

Navegación.

- **L.a_primero**(): Coloca el cursor en el primer elemento de L.
  - **PRE**: $\exists L \wedge L\neq\{NULL\}$
  - **POST**: Cursor=1
- **L.a_ultimo**(): Coloca el cursor en el ultimo elemento de L.
  - **PRE**: $\exists L \wedge L\neq\{NULL\}$
  - **POST**: Cursor=n
- **L.a_anterior**(): Señala el anterior.
  - **PRE**: $\exists L \wedge L\neq\{NULL\} \wedge i=Cursor$
  - **POST**: if(i=1){Cursor=i} else {Cursor=i-1}
- **L.a_siguiente**(): Señala el siguiente.
  - **PRE**: $\exists L \wedge L\neq\{NULL\} \wedge i=Cursor$
  - **POST**: if(i=n){Cursor=i}else{Cursor=i+1}
- **L.a_iesimo**(k): Señala el késimo.
  - **PRE**: $\exists L \wedge L\neq\{NULL\} \wedge i=Cursor$
  - **POST**: if($k<1 \vee k>n$){Cursor=k} else {Cursor=i}
- **L.a_clave**(k): Señala el de clave k.
  - **PRE**: $\exists L \wedge L\neq\{NULL\} \wedge i=Cursor$

- **POST**: if(existe(k)){Cursor=posicion(k)} else {Cursor=i}

Inserción y borrado.

- **L.inserte_antes**(e): inserta e antes del actual.
  - **PRE**: $\exists L \wedge L \neq \{NULL\} \wedge L=\{PRF,a,POF\} \wedge actual(L)=a \wedge tamaño(PRF)=Cursor-1 \wedge i=Cursor$
  - **POST**: $L=\{PRF, e,a, POF\} \wedge Cursor=i$
  - **PRE**: $\exists L \wedge L=\{NULL\}$
  - **POST**: $L=\{e\} \wedge Cursor=1$
- **L.inserte_despues**(e): inserta e despues del actual.
  - **PRE**: $\exists L \wedge L \neq \{NULL\} \wedge L=\{PRF,a,POF\} \wedge actual(L)=a$
    $\wedge tamaño(PRF)=Cursor-1 \wedge i=Cursor$
  - **POST**: $L=\{PRF,a,e,POF\} \wedge Cursor =i+1$
  - **PRE**: $\exists L \wedge L=\{NULL\}$
  - **POST**: $L=\{e\} \wedge Cursor=1$
- **L.borre**(): borra el actual.
  - **PRE**: $\exists L \wedge L \neq \{NULL\} \wedge L=\{PRF,a,POF\} \wedge a=actual(L)$
    $\wedge tamaño(PRF)=Cursor-1 \wedge i=Cursor$
  - **POST**: $L=\{PRF, POF\} \wedge$ if $(PRF \neq \{NULL\})$ Cursor=i-1
    else if$(POF \neq \{NULL\})$ Cursor=i else Cursor=0

Actualización.

- **L.actualize**(e): Actualiza elemento actual.
  - **PRE**: $\exists L \wedge L \neq \{NULL\} \wedge L=\{PRF,a,POF\} \wedge a=actual(L) \wedge tamaño(PRF)=Cursor-1$
    $\wedge clave(a)=clave(e)$
  - **POST**: $L=\{PRF,e,POF\}$

Consultores.

- **e=L.actual**(): recupera actual.
  - **PRE**: $\exists L \wedge L \neq \{NULL\} \wedge L=\{PRF,a,POF\} \wedge tamaño(PRF)=Cursor-1$
  - **POST**: e=a
- **t=L.posicion**(e): posicion de e.
  - **PRE**: $\exists L \wedge L \neq \{NULL\} \wedge L=\{PRF,e,POF\}$
  - **POST**: t=tamaño(PRF)+1
- **n=L.tamaño(): tamaño de la lista**.
  - **PRE**: $\exists L \wedge L=\{e,POF\}$
  - **POST**: if(L={NULL})n=0
    else n=1+tamaño(POF)
- **b=l.existe**(k): existe el elemento de clave k?.
  - **PRE**: $\exists L \wedge L=\{e,POF\}$
  - **POST**: if(clave(e)=k)b=true
    else **existe**(POF,k)
  - **PRE**: $\exists L \wedge L=\{NULL\}$
  - **POST**: b=false

Las pilas y las colas son casos especiales de las listas; Su especificación se diferencia de las listas sólo en las funciones de inserción y borrado (métodos de inserción y eliminación de componentes de la lista).

2.1 PILAS

Inserción y borrado.

- **L.push**(e:stelement): inserta e después ultimo elemento insertado.
  - **PRE**: $\exists L \wedge L \neq \{NULL\} \wedge L=\{PRF\} \wedge$ tamaño(L)=n
  - **POST**: $L=\{PRF,e\} \wedge$ tamaño(L)=n + 1
  - **PRE**: $\exists L \wedge L=\{NULL\}$
  - **POST**: $L=\{e\} \wedge n=1$

- **L.pop**(): borra el elemento recientemente insertado.
  - **PRE**: $\exists L \wedge L \neq \{NULL\} \wedge L=\{PRF, e\} \wedge$ tamaño(L)=n
  - **POST**: $L=\{PRF\} \wedge$ tamaño(L)=n - 1

2.2 COLAS

Inserción y borrado.

- **L.enqueue** (e:stelement): inserta e después ultimo elemento insertado.
  - **PRE**: $\exists L \wedge L \neq \{NULL\} \wedge L=\{PRF\} \wedge$ tamaño(L)=n
  - **POST**: $L=\{PRF,e\} \wedge$ tamaño(L)=n + 1
  - **PRE**: $\exists L \wedge L=\{NULL\}$
  - **POST**: $L=\{e\} \wedge n=1$

- **L.serve**(): borra el elemento menos recientemente insertado.
  - **PRE**: $\exists L \wedge L \neq \{NULL\} \wedge L=\{e, POF\} \wedge$ tamaño(L)=n
  - **POST**: $L=\{POF\} \wedge$ tamaño(L)=n - 1

3 PROGRAMACIÓN CON LISTAS EN C++

Usando las funciones dadas en la especificación, pueden llevarse a cabo programas de manejo de listas que permiten aprovechar toda la funcionalidad ofrecida por esta estructura de datos.

Ejemplo 3.1:

A continuación se presenta la función leer_estudiante que se encarga de leer los datos de un estudiante por teclado y devolver el estudiante con sus datos ya ingresados.

```
estudiante leer_estudiante()
{

      estudiante e;

      float t;
      printf("Escriba el carne:  ");
      scanf("%i",&(e.aCarne));

      printf("Escriba el nombre: ");
      scanf("%s",(e.aNombre));

      printf("Escriba la nota: ");
      scanf("%f",&t);
      e.aNota=t;

      printf("Escriba el año de ingreso: ");
      scanf("%i",&(e.aAnno_ing));

      printf("\n\n");

      return e;
}
```

Ejemplo 3.2:

A continuación se presenta la función ver_estudiante que se encarga de mostrar los datos de un estudiante en particular en pantalla.

```
void ver_estudiante(estudiante e)
{
     printf("\nClave: %i\n",e.key());
     printf("Nombre: %s\n",e.aNombre);
     printf("Nota: %f\n",e.aNota);
     printf("\n\n");
}
```

Ejemplo 3.3

A continuación se presenta la función contar_nota que se encarga de contar el numero de estudiantes con una nota dada y devuelve este numero.

```
int contar_nota(lista l, float nota)
{
     int count=0;

        for(int i=0; i<l.tamanno(); i++)
```

```
        {
            if(l.primero[i].aNota==nota)
            count++;
        }

      return count;
}
```

Ejemplo 3.4:

La función concatenar añade los elementos de la lista l2 al final de la lista l1 (concatena las listas).

```
void concatenar(lista *l1, lista *l2)
{

      l1->a_ultimo();
      l2->a_primero();

      for(int i=0; i<(l2->tamanno()); i++)
      {
            l1->inserte_despues(l2->actual());
            l2->a_siguiente();
      }

}
```

Ejemplo 3.5:

La siguiente función crea una copia de una lista dada. Para hacerlo se recorre la lista original y por cada elemento de esta se va creando un nuevo elemento en la lista de destino. Tanto la lista que desea copiarse (lista fuente) como la lista donde será copiada (lista destino) se pasan como argumentos a la función por medio de punteros. Si la lista destino no es vacía, entra en ejecución el código del primer if que la limpia (borra todos sus elementos)

```
void copiar_lista(lista *f, lista *d)
{

      int t=f->tamanno();
      int td=d->tamanno();
      f->a_primero();

      if(d->tamanno()>0)
      {
           d->a_primero();
```

```
            for(int i=0; i<td; i++)
            {
                  d->borre();
            }

      }

      for(int i=0; i<t; i++)
      {
            d->inserte_despues(f->actual());
            f->a_siguiente();
            d->a_siguiente();
      }
}
```

A Referencias

[DS1984] Stubbs. Daniel F. & Webre, Neil W., "Data Structures: with Abstract Data Types and Pascal", Brooks/Cole Publishing Company, 1984.
1: Secciones 3.2, 3.3, 4.2.1

Curso de C. Gorka Urrutia. El capítulo 9 da una explicación general de los punteros. Disponible para bajarse en la página del curso. También se encuentra en www.elrincondelc.com.

B Taller

a) Describa la pre y post condicion de las siguientes funciones y luego indique en C++, su implementación con base en las funciones especificadas para la lista.

- **a_clave(L,k)**: Señala elemento con clave k.
- **V=es_vacia(L)**:.Indica si la lista es vacia o no (true, false).
- **recorrer(L,f(e))**: recorre lista aplicando f(e) a cada elemento.
- **Invierta_lista(L)**: invierte el orden de los elementos de L.
- **Borrar_n_primeros(n)**: borra los n primeros elementos de la lista.