Tema 3

TDAs contenedores básicos

Objetivos Generales

- Entender el concepto de contenedor
- Conocer los principales tipos de contenedores básicos
- Resolver problemas usando contenedores
- Entender el concepto de iterador y ser capaz de usarlos.

Introducción a los contenedores

Contenedor:

"Embalaje metálico grande y recuperable, de tipos y dimensiones normalizados internacionalmente y con dispositivos para facilitar su manejo."

(Diccionario RAE)

Introducción a los contenedores

Un contenedor es una colección de objetos dotado de un conjunto de métodos para gestionarlos (acceder a ellos, eliminarlos, añadir nuevos objetos, etc.).

Los contenedores se pueden clasificar según distintos criterios, pero la fundamental es la forma de organización:

Lineal, jerárquica, interconexión total

Introducción a los contenedores

También se diferencia por las formas de acceder a los componentes:

- secuencial
- directa
- por clave
  (también conocidos como asociativos)

Contenedores secuenciales

Organizan los datos en orden lineal:

primer elemento, segundo, ...

Además, quedan identificados por la posición que ocupan en el contenedor.

Vector dinámico (vector)

Lista (list)

Pila (stack)

Cola (queue)

Contenedores directos

Garantizan acceso a cualquier componente en tiempo constante:

Vector dinámico (vector)

Contenedores asociativos

Gestionan los datos mediante claves que los identifican (por ejemplo, DNI) y permiten su recuperación eficiente a partir de ellas.

Conjunto (set)

Bolsa (multiset)

Diccionario (map)

El contenedor Pila (*stack*)

Estructura de datos, representada como una sucesión de objetos, que limita la forma en que éstos elementos se añaden a ella, así como la manera de abandonarla.

Concretamente, una pila sólo permite introducir y borrar elementos por un extremo de la secuencia, denominado *tope (top)* (Secuencias LIFO).

El contenedor Cola (*queue*)

Contenedor, representado como una sucesión de objetos que limita la forma en que éstos se añaden a ella, al igual la manera de abandonarla.

Así, una cola sólo permite introducir objetos por un extremo de la sucesión, *final*, y la recuperación por el opuesto, *frente*. Los elementos se ubican en estricto orden de llegada. (Secuencia FIFO).

El contenedor Cola con Prioridad (*priority_queue*)

Tipo especial de cola en la que se siguen cumpliendo las restricciones en cuanto a los lugares por donde se realiza la inserción y acceso de los elementos.

Pero en este caso los elementos no se disponen según su orden de llegada, sino de acuerdo a algún tipo de prioridad.

El contenedor Conjunto (*set*)

Colección de valores que no se repiten y que permite conocer de manera eficiente si un valor está contenido o no en el conjunto.

El contenedor Bolsa (*multiset*)

Análogo a un conjunto, pero permitiendo valores repetidos.

El contenedor Diccionario (*map*)

Estructura de almacenamiento que implementa una relación ``clave-valor".

Permite almacenar valores identificados por claves únicas, y recuperarlos previamente mediante éstas.

El contenedor Vector Dinámico (*vector*)

Generalización de un vector o array que almacena una colección de elementos del mismo tipo.

Los elementos se acceden de manera directa por medio de un índice, en el rango de 0 a n-1, siendo n el tamaño del vector.

Dicho tamaño puede aumentarse o decrementarse según las necesidades de manera dinámica.

El contenedor Lista (*list*)

Contenedor que almacena sus elementos en forma de sucesión, permitiendo, por tanto, el acceso secuencial a ellos.

Cada elemento establece quién es el siguiente de la sucesión.

Abstracción por iteración: el concepto de iterador

Son generalizaciones de punteros: objetos que “señalan” a otros objetos

(pero no son punteros!!).

Se utilizan para iterar (moverse) sobre un rango de objetos de manera independiente del tipo de contenedor donde estén incluidos.

Permitirán acceder a elementos del contenedor, modificarlos, borrarlos o insertar nuevos.

Iteradores en la STL

Las clases de la STL poseen cuatro clases de iteradores dependiendo de:

- La forma de recorrer, o
- Si permite sólo lecturas (L), o insertar otros nuevos o modificar ya existentes (E):

iterator: hacia adelante (L, E).

const_iterator: hacia adelante (L).

reverse_iterator: hacia atrás (L, E).

const_reverse_iterator: hacia atrás (L).

Especificación iteradores STL

Especificaremos las operaciones de los iteradores comunes a los contenedores *set*, *multiset*, *map*, *vector* y *list*.

Para especificar los TDAs iterator y const_iterator, utilizaremos genéricamente el TDA contenedor.

Especificación iteradores STL (I)

Declaración de iteradores:

```
contenedor<T>::iterator miIterador;
contenedor<T>::const_iterator
miIteradorConst;
```

Ejemplo:

```
list<double>::iterator iterListaDouble;
map<String,int>::const_iterator iterDiccConst;
```

Especificación iteradores STL (II)

/**

TDA iterator::iterator, *, ++, --, ==, !=, ~iterator

El TDA iterator está asociado a un contenedor y señala los elementos de T contenidos en él.

Permite recorrer el contenedor con posible modificación del mismo, por lo que sólo se puede usar con contenedores no constantes.

Es mutable.

Especificación iteradores STL (III)

TDA const_iterator::const_iterator, *, ++, --, ==, !=, ~const_iterator

El TDA const_iterador está asociado a un contenedor y señala los elementos de T contenidos en él.

Permite recorrer el contenedor pero no modificarlo por lo que se puede usar tanto con contenedores constantes como no constantes.

Es inmutable. */

Especificación iteradores STL (IV)

```
/**
  @brief Constructor primitivo.
  Crea un iterador nulo.
*/
    const_iterator();
/**
  @brief Constructor de copia.
  @param i: iterador que se copia.
  Crea un iterador copia de i.
*/
    const_iterator(const const_iterator & i);
```

Especificación iteradores STL (V)

```
/**
  @brief Obtener el elemento al que apunta el iterador.
  @pre El iterador NO está al final del recorrido.
       Es distinto de end().
  @return elemento del contenedor en la posición apuntado
  por el iterador.
*/
    T operator*() const;
```

Especificación iteradores STL (VI)

```
/**
  @brief Operador de incremento.
  @pre El iterador NO está al final del recorrido.
            Es distinto de end().
  @return Referencia al iterador receptor.
  Hace que el iterador apunte a la posición siguiente en el
  contenedor y lo devuelve.
*/
    const_iterator & operator++();
```

Especificación iteradores STL (VII)

```
/**
  @brief Operador de decremento.
  @pre El iterador NO está al principio del recorrido.
       Es distinto de begin().
  @return Referencia al iterador receptor.
  Hace que el iterador apunte a la posición anterior
       en el contenedor y lo devuelve.
*/
     const_iterator & operator--();
```

Especificación iteradores STL (VIII)

```
/**
  @brief Comparación de igualdad.
  @param i: segundo iterador en la comparación.
  @return true, si el receptor e 'i' tienen el mismo valor.
                false, en otro caso.
*/
    bool operator==(const const_iterator & i);
```

Especificación iteradores STL (IX)

```
/**
   @brief Comparación de desigualdad.
   @param i: segundo iterador en la comparación.
   @return true, si el receptor e 'i' tienen un valor  distinto.
                 false, en otro caso.
*/
      bool operator!=(const const_iterator & i);
```

Especificación iteradores STL (X)

Ejemplo*:*

```
template <class T1, class T2>
void ImprimirContenedor (const T1<T2>& contenedor)
   {
     T1::const_iterator iter;
     for (iter  = contenedor.begin();  iter != contenedor.end();
          iter++)
                cout << *iter << endl;
  }
```

Especificación TDA Pila (stack) (II)

*U*na “pila” de libros, en donde éstos se “apilan" unos sobre otros y sólo se puede coger el que está en la parte superior de ella y poner un nuevo libro encima del colocado en dicha parte superior.

Una “pila” de bandejas en los comedores universitarios. Los comensales cojen la que está en la parte superior y el personal, una vez limpias, las va colocando encima de la última colocada.

En la STL, este TDA recibe el nombre de **stack**.

Especificación TDA Pila (*stack*) (III)

*/** stack<T>*

TDA stack::stack, empty, clear, swap, top, push, pop, size, ~stack

Cada objeto del TDA pila, modela una pila de elementos de la clase T.

Una pila es una secuencia de elementos donde las inserciones y borrados tienen lugar en un extremo denominado "tope". Son contenedores del tipo LIFO (Last In, First Out).

Son objetos mutables. Residen en memoria dinámica.

**/*

Especificacion TDA Pila (*stack*) (II)

```
/**
    @brief Constructor primitivo.
    Crea una pila vacía.
 */
     stack();
 /**
    @brief Constructor de copia.
    @param p: pila que se copia.
    Crea una pila que es copia de p.
 */
        stack(const stack<T> & p);
```

Especificacion TDA Pila (*stack*) (III)

```
/**
    @brief Constructor primitivo.
    Crea una pila vacía.
 */
     stack();
 /**
    @brief Constructor de copia.
    @param p: pila que se copia.
    Crea una pila que es copia de p.
 */
      stack(const stack<T> & p);
```

Especificacion TDA Pila (*stack*) (III)

```
/**
    @brief Informa si la pila está vacía.
    @return true, si la pila está vacía.
                false, en otro caso.
 */
        bool empty() const;
/**
   @brief Borra todos los elementos del contenedor.
   Deja el contenedor completamente vacío.
*/
        void clear();
```

Especificacion TDA Pila (stack) (IV)

```
/**
  @brief Intercambia el contenido del receptor y del argumento.
  @param p: pila a intercambiar con el receptor. ES MODIFICADO.
  Este método asigna el contenido del
      receptor al del parámetro y el del  parámetro al del receptor.
*/
      void swap (stack<T> & p);
```

Especificacion TDA Pila (stack) (V)

```
/**
   @brief Acceso al elemento en lo alto de la pila.
   @pre El receptor no puede estar vacío: !empty().
   @return Referencia al elemento en el tope
*/
      T& top ();
/**
   @brief Acceso al elemento en el tope de la pila.
   @pre El receptor no puede estar vacío: !empty().
   @return Referencia constante al tope de la pila.
*/
      const T& top () const;
```

Especificacion TDA Pila (stack) (VI)

```
/**
   @brief Deposita un elemento en la pila.
   @param elem: Elemento que se inserta.
   Inserta un nuevo elemento en el tope de la pila.
*/
      void push(const T & elem);
/**
   @brief Quita un elemento de la pila.
   @pre El receptor no puede estar vacío: !empty().
   Elimina el elemento en el tope de la pila.
*/
      void pop();
```

Especificacion TDA Pila (stack)(VII)

```
/**
   @brief Obtiene el número de elementos.
   @return Número de elementos de la pila.
*/
      size_type size() const;
/**
   @brief Destructor.
   @post El receptor es MODIFICADO.
   El receptor es destruido liberando los recursos.
*/
        ~stack();
```

Otras operaciones: ==, !=, <, >, <=, >= , =.

Declaración de objetos Pila

```
#include <stack>
using namespace std;

stack<int> pilaEnteros;
stack<String> pilaCadenas;
stack<MiTipo> pilaMiTipo;
stack<queue<double>> pilaDeColas;
```

Ejemplos de uso del TDA Pila (I)

```
#include <iostream>
#include <stack>
using namespace std;
int main()
{
    stack<int> st;
    // Se introducen tres elementos en la pila.
    st.push(1);     st.push(2);     st.push(3);

    // Se sacan e imprimen dos elementos.
    cout << st.top() << ' ';     st.pop();
    cout << st.top() << ' ';     st.pop();

    // Se modifica el del tope.
    st.top() = 77;
```

Ejemplos de uso del TDA Pila (I)

```
    // Se introducen dos nuevos elementos.
    st.push(4);
    st.push(5);

    // Se elimina el del tope sin procesarlo.
    st.pop();

    // Se sacan de la pila el resto de elementos.
    while (!st.empty()) {
        cout << st.top() << ' ';
        st.pop();
    }
    cout << endl;
}
```

Ejemplos de uso del TDA Pila (II)

Evaluación de una expresión postfija.

```
/**
 @param e: expresion en postfijo (cada elemento  del vector
  se corresponde con un elemento de la expresion).
 @return  el resultado de evaluar la expresion
*/
int evalua(String e)
{
  stack<int> v;
  int i, izq,dcha;
```

Ejemplos de uso del TDA Pila (II)

```
  for (i=0; i!=e.size(); ++i) {
 // recorrido de los caracteres del string.
          if ((e[i]>='0') && (e[i]<='9'))
          v.push(e[i]-'0');
      else    {
         dcha = v.top(); v.pop();
         izq = v.top(); v.pop();
         switch (e[i]) {
                case '+': v.push(izq+dcha);  break;
                case '-':  v.push(dcha-izq);  break;
          case '*':v.push(izq*dcha);   break;
          case  '/': v.push(dcha/izq);   break;
           } // switch
        } // else
      }  // for
  return v.top();
} // fin de la función
```

El TDA Cola (*queue*)

Una cola es una secuencia de elementos que permite el acceso a los mismos sólo por los dos extremos:

por uno se insertan nuevos objetos (*final*, *end*) y

por el otro se accede a ellos (*frente*, *front*).

El resto de elementos no está accesible de manera directa.

El TDA Cola (queue) - Ejemplos

Una *cola de clientes en una caja de un supermercado esperando a ser cobrados. Los clientes acceden por el final de la cola y son atendidos por el frente. El cajero sólo atiende a aquel cliente que esté en el frente.*

Una cola de impresión que gestiona la impresión de trabajos en una impresora.

La cola que se forma en el banco para realizar alguna operación.

*En la STL, el TDA Cola tiene el nombre de **queue**.*

Especificación del TDA Cola (I)

/**

queue<T>

TDA queue::queue, empty, clear, front, push, pop, size, swap, ~queue

Cada objeto del TDA cola, modela una cola de elementos de la clase T.

Una cola es un tipo particular de secuencia en la que los elementos se insertan por un extremo (final) y se consultan y suprimen por el otro (frente). Son secuencias del tipo FIFO (First In, First Out).

Son objetos mutables. Residen en memoria dinámica.

*/

Especificación del TDA Cola (II)

```
/**
   @brief Constructor primitivo.
   Crea una cola vacía
 */
        queue();
/**
   @brief Constructor de copia.
   @param c: cola que se copia.
   Crea una cola que es copia de c.
*/
        queue(const queue<T> & c);
```

Especificación del TDA Cola (III)

```
/**
  @brief Informa si la cola está vacía.
  @return true, si la cola está vacía.
          false, en otro caso.
*/
    bool empty() const;
/**
  @brief Borra todos los elementos del contenedor.
  Deja el contenedor completamente vacío.
*/
    void clear();
```

Especificación del TDA Cola (IV)

```
/**
  @brief Acceso al elemento al principio de la cola.
  @pre El receptor no puede estar vacío.
  @return Referencia al elemento en el frente de la cola.
*/
      T & front ();
/**
    @brief Acceso al elemento al principio de la cola.
    @pre El receptor no puede estar vacío.
    @return Referencia constante al  frente de la cola.
*/
      const T & front () const;
```

Especificación del TDA Cola (V)

```
/**
   @brief Añade un elemento en la cola.
   @param elem: Elemento que se inserta.
   Inserta un nuevo elemento al end de la cola.
*/
     void push (const T & elem);
/**
   @brief Quita un elemento de la cola.
   @pre El receptor no puede estar vacío.
   Elimina el elemento en el frente de la cola.
*/
     void pop();
```

Especificación del TDA Cola (VI)

```
/**
   @brief Obtiene el número de elementos en la cola.
   @return número de elementos incluidos en la cola.
 */
     size_type size() const;
/**
  @brief Intercambia el contenido del receptor y argumento.
  @param c: cola a intercambiar con el receptor.
            ES MODIFICADO.
  Asigna el contenido del receptor al del parámetro y
viceversa.
 */
     void swap (queue<T> & c);
```

Especificación del TDA Cola (VII)

```
/**
   @brief Destructor.
   @post El receptor es MODIFICADO.

   El receptor es destruido liberando todos los
   recursos que usaba.
*/
    ~queue();
```

Otras operaciones:

==, !=, <, >, <=, >= , =

Ejemplos de uso del TDA Cola

Declaración de objetos:

```
#include <queue>
using namespace std;

queue<int> colaEnteros;
queue<String> colaCadenas;
queue<MiTipo> colaMiTipo;
queue<list<double>> colaDeListas;
```

Ejemplos de uso del TDA Cola (II)

```
#include <iostream>
#include <queue>
#include <string>
using namespace std;

int main()
{
    queue<string> q;

    // Inserción de tres elmentos en la cola.
    q.push("Tenemos ");   q.push("más");  q.push("de");
```

Ejemplos de uso del TDA Cola (III)

```
// Impresión de dos elementos.
cout << q.front();     q.pop();
cout << q.front();     q.pop();

// Inserción de dos nuevos elementos.
q.push("cuatro");
q.push("elementos");

// Pasamos del elemento del frente.
q.pop();
```

Ejemplos de uso del TDA Cola (IV)

```
    // Impresión de dos elementos.
    cout << q.front();    q.pop();
    cout << q.front() << endl;    q.pop();

    // impresión del número de elementos en la cola.
    cout << "Número de elementos: " << q.size()  << endl;
}
```

El TDA Cola con prioridad (priority_queue)

Una cola con prioridad es una secuencia que permite el acceso sólo por los extremos:

Uno de inserción nuevos objetos (final, end)

Otro se accede a ellos (frente, front).

La principal característica es que al insertar los elementos estos se *sitúan en la cola según una cierta prioridad.*

El TDA Cola con prioridad (priority_queue)

Ejemplos:

La cola de espera de unas emergencias sanitarias. Prioridad según la gravedad del enfermo.

Una cola de impresión de una impresora. Prioridad según el tipo de usuario o la longitud del mismo.

En la STL, *priority_queue*.

Especificación del TDA Cola con prioridad (I)

/** priority_queue<T,Contenedor,Comp>

TDA priority_queue::priority_queue, empty, clear, top, push, pop, size, swap, ~priority_queue.

Cada objeto del TDA cola con prioridad, modela una cola con prioridad de elementos de la clase T.

Una cola con prioridad es un tipo particular de secuencia en la que los elementos se insertan por un extremo (final) y se consultan y suprimen por el otro (frente).

Los elementos se disponen en ella ordenados según su prioridad.

Son objetos mutables. Residen en memoria dinámica.

*/

Especificación del TDA Cola con prioridad (II)

```
/**
    @brief Constructor primitivo.
    @doc  Crea una cola con prioridad vaca.
 */
     priority_queue();
/**
    @brief Constructor de copia.
    @param c: cola con prioridad que se copia.
    @doc Crea una cola con prioridad que es copia de c.
*/
     priority_queue(const priority_queue<T> & c);
```

Especificación del TDA Cola con prioridad (III)

```
/**
   @brief Informa si la cola con prioridad está vacía.
   @return true, si la cola con prioridad est vacía.
                 false, en otro caso.
*/
     bool empty() const;
/**
   @brief Vacía la cola.
   Elimina todos los elementos de la cola, dejándola vacía.
*/
     void clear();
```

Especificación del TDA Cola con prioridad (IV)

```
  /**
    @brief Acceso al elemento del frente.
    @pre El receptor no puede estar vacío.
    @return Referencia al elemento en el frente.
  */
     T & top ();
  /**
    @brief Acceso al elemento del frente.
    @pre El receptor no puede estar vacío.
    @return Referencia constante al elemento del frente.
  */
     const T & top () const;
```

Especificación del TDA Cola con prioridad (V)

```
/**
   @brief Añade un elemento en la cola con prioridad.
   @param elem: Elemento que se inserta.
   @pre Se requiere que exista una operación “<”.
   Inserta un nuevo elemento en la cola según
        su prioridad, quedando ordenados de mayor a menor.
        Eficiencia: logarítmica.
*/
    void push (const T & elem);
```

Especificación del TDA Cola con prioridad (VI)

```
/**
    @brief Elimina un elemento de la cola con prioridad.
    @pre El receptor no puede estar vacío.
    Elimina el elemento en el frente de la cola
                con prioridad. Eficiencia: logarítmica.
*/
    void pop();
```

Especificación del TDA Cola con prioridad (VI)

```
/**
    @brief Obtiene el número de elementos en la cola.
    @return número de elementos en la cola  con prioridad.
*/
     size_type size() const;
/**
  @brief Intercambia contenidos del receptor y del arg.
  @param c: cola con pri. a intercambiar.  ES MODIFICADO.
  Asigna el contenido del receptor al del parámetro y
  viceversa.
*/
        void swap (priority_queue<T> & p);
```

Especificación del TDA Cola con prioridad (VII)

```
/**
   @brief Destructor.
   @post El receptor es MODIFICADO.

   El receptor es destruido liberando todos los
   recursos que usaba.
*/
~priority_queue();
```

Otras operaciones: ==, !=, <, >, <=, >= , =

Ejemplos de uso TDA Cola con prioridad (I)

Declaración de objetos:

```
#include <priority_queue>
using namespace std;

priority_queue<int> colaPrioridadEnteros;
priority_queue<String> colaPrioridadCadenas;
priority_queue<MiTipo> colaPrioridadMiTipo;
priority_queue<queue<double>> colaPriorDeColas;
```

Ejemplos de uso TDA Cola con prioridad (II)

```
#include <iostream>
#include <queue>
using namespace std;

int main()
{
    priority_queue<float> q;

    // Inserción de tres elementos en la cola con prioridad.
    q.push(66.6);     q.push(22.2);     q.push(44.4);
```

Ejemplos de uso TDA Cola con prioridad (III)

```
// Impresión de dos elmentos.
cout << q.top() << ' ';     q.pop();
cout << q.top() << endl;     q.pop();

// Inserción de tres más.
q.push(11.1);
q.push(55.5);
q.push(33.3);

// Eliminación de uno de ellos.
q.pop();
```

Ejemplos de uso TDA Cola con prioridad (IV)

```
    // Sacamos e imprimimos el resto.
    while (!q.empty()) {
        cout << q.top() << ' ';
        q.pop();
    }
    cout << endl;
}
```

Ejemplos de uso TDA Cola con prioridad (V)

Ordenación de un vector mediante una cola con prioridad

```
#include <priority_queue>

void ordenar(int *v, int n)
{
  priority_queue<int> colaPrioridad;
  int i;
  for (i=0; i< n; i++)
    colaPrioridad.push(v[i]);
```

Ejemplos de uso TDA Cola con prioridad (VI)

```
  for (i=0; i< n; i++)
   {
     v[i]= colaPrioridad.top();
     colaPrioridad.pop();
   }
}
```

El TDA Conjunto (set)

Un conjunto es un contenedor de valores únicos, llamados claves o miembros del conjunto, que se almacenan de manera ordenada.

El TDA Conjunto está basado en el concepto matemático de conjunto.

En la STL, recibe el nombre de *set.*

El TDA Conjunto (set) (II)

Ejemplo:

Un corrector ortográfico de un procesador de textos puede utilizar un conjunto para almacenar todas las palabras que considera correctas ortográficamente hablando.

Especificación del TDA Conjunto (set) (I)

/**

set<T,Comp>

TDA set:: set, empty, clear, size, count, find, begin, end, lower_bound, upper_bound, insert, erase, swap, ~set

Cada objeto del TDA Conjunto, modela un conjunto de elementos de la clase T.

Un conjunto es un contenedor que almacena elementos de manera ordenada que no están repetidos.

Son objetos mutables. Residen en memoria dinámica.

*/

Especificación del TDA Conjunto (set) (II)

```
/**
    @brief Constructor primitivo.
    Crea un conjunto vaco.
 */
    set();
/**
    @brief Constructor de copia.
    @param c: conjunto que se copia.
    Crea un nuevo conjunto que es copia de c.
*/
    set(const set<T> & c);
```

Especificación del TDA Conjunto (set) (III)

```
/**
    @brief Constructor de copia de un rango.
    @param inicio: apunta al elemento inicial a copiar.
    @param final: apunta al elmento final a copiar.
    Crea un nuevo conjunto que es copia del rango
        [inicio,final).
 */
    set(iterator inicio, iterator final);
```

Especificación del TDA Conjunto (set) (IV)

```
/**
   @brief Informa si el conjunto está vacío.
   @return true, si el conjunto está vacío.
           false, en otro caso.
*/
    bool empty() const;
/**
   @brief Borra todos los elementos del contenedor.
   Deja el contenedor completamente vacío.
       Eficiencia: lineal.
*/
    void clear();
```

Especificación del TDA Conjunto (set) (V)

```
/**
   @brief Obtiene el número de elementos.
   @return Número de elementos del conjunto.
*/
    size_type size() const;
/**
   @brief Busca un elemento en el conjunto.
   @param clave: elemento a buscar.
   @return 1 si el elemento está en el conjuto y 0 si no.
   Determina si un elemento está o no en el
       conjunto receptor. Eficiencia: Lineal.
*/
int count(const T & clave) const;
```

Especificación del TDA Conjunto (set) (VI)

```
/**
   @brief Busca un elemento en el conjunto.
   @param clave: elemento a buscar.
   @return Un iterador que apunta al elemento dentro
           del conjunto en caso de que se encuentre
           en él, o end() en caso contrario.
           Eficiencia: Logartímica.
*/
   iterator find(const T & clave);
```

Especificación del TDA Conjunto (set) (VI)

```
/**
  @brief Determina la primera posición donde se
      insertará un elemento.
  @para clave: clave cuya ubicación se desea encontrar.
  @return Un iterador que señala a la primera posición
        donde una copia del argumento se insertará según
        el criterio de ordenación.
  Según el critero de inserción que se use,
        determina la primera posición donde se insertará
        una clave. Eficiencia: logarítmica.
*/
      iterator lower_bound (const T & clave);
```

Especificación del TDA Conjunto (set) (VII)

```
/**
 @brief Determina la última posición donde se
        insertará un elemento.
 @para clave: clave cuya ubicación se desea encontrar.
 @return Un iterador que señala a la última posición
         donde una copia del argumento se insertará según
         el criterio de ordenación.
 Según el critero de inserción que se use,
         determina la última posición donde se insertará
         una clave. Eficiencia: logarítmica.
*/
    iterator upper_bound (const T & clave);
```

Especificación del TDA Conjunto (set) (VIII)

```
/**
    @brief Busca un elemento en el conjunto.
    @param clave: elemento a buscar.
    @return Un iterador constante que apunta al elemento
            dentro del conjunto en caso de que se
            encuentre en él, o end() en caso contrario.
            Eficiencia: logarítmica.
 */
    const_iterator find(const T & clave);
```

Especificación del TDA Conjunto (set) (X)

```
/**
    @brief Borra un elemento del conjunto.
    @param clave: elemento a eliminar.
    @return 1 si el elemento estaba en el conjunto y
            ha sido eliminado, 0 en caso contrario.
   Al eliminarse el elemento, el tamaño se
        decrementa en una unidad.
        Eficiencia: logarítmica.
 */
     int erase(const T & clave);
```

Especificación del TDA Conjunto (set) (XI)

```
/**
   @brief Borra un elemento del conjunto.
   @param pos: señala al elemento a borrar.
   @pre El conjunto no est vaco.
   Elimina el elemento y el tamaño del
        conjunto se decrementa en una unidad.
*/
    void erase(iterator pos);
```

Especificación del TDA Conjunto (set) (XII)

```
/**
   @brief Borra elementos del rango [primero,ultimo).
   @param primero: primer elemento del rango.
          ultimo: ltimo elmento del rango.
   @pre El conjunto no est vacío.
   Elimina los elementos incluidos en el rango
       pasado como argumento. El tamaño del
       conjunto se decrementa según los
       elementos del rango.
*/
    void erase(iterator primero, iterator ultimo);
```

Especificación del TDA Conjunto (set) (XIII)

```
/**
    @brief Devuelve un iterador señalando al primer
        elemento del conjunto.
    @return Un iterador apuntando al primer elemento.
*/
      iterator begin();
/**
  @brief Devuelve un iterador constante señalando al
      primer elemento del conjunto.
  @return Iterador constante al primer elemento.
*/
      const_iterator begin() const;
```

Especificación del TDA Conjunto (set) (XIV)

```
/**
   @brief Devuelve un iterador sealando al último
          elemento del conjunto.
   @return Un iterador apuntando al último elemento.
 */
    iterator end();
/**
  @brief Devuelve un iterador constante al último
    elemento del conjunto.
  @return Un iterador constante al úlltimo elemento.
 */
    const_iterator end() const;
```

Especificación del TDA Conjunto (set) (XV)

```
/**
 @brief Intercambia el contenido del receptor y  del argumento.
 @param s: set a intercambiar con el receptor.   ES MODIFICADO.
 Asigna el contenido del receptor al del parmetro y viceversa.
*/
      void swap (set<T> & s);
/**
   @brief Destructor.
   @post El receptor es MODIFICADO.
   El receptor es destruido y se liberan recursos.
*/
~set();
          Otras operaciones: ==, !=, <, >, <=, >= , =.
```

Ejemplos uso del TDA Conjunto (set) (I)

```
#include <set>
using namespace std;

set<int> conjuntoEnteros; // Ordenación creciente.
set<double, greater(double)> conjuntoReales;
set<long, less(long)> conjuntoMiTipo;

struct ltstr {
  bool operator()(const char* s1,  const char* s2) const
  {   return strcmp(s1, s2) < 0;  }
};
```

Ejemplos uso del TDA Conjunto (set) (II)

```
#include <iostream>
#include <set>
using namespace std;
int main()
{
    typedef set<int,greater<int> > IntSet;
    IntSet coll1;        // Conjunto vacío de enteros.
    // Inserción de elementos ``aleatoriamente''.
    coll1.insert(4);   coll1.insert(3);   coll1.insert(5);
    coll1.insert(1);   coll1.insert(6);   coll1.insert(2);
    coll1.insert(5);
```

Ejemplos uso del TDA Conjunto (set) (III)

```
// Iteramos sobre todos los elementos.
IntSet::iterator pos;
for (pos = coll1.begin(); pos != coll1.end(); ++pos)
    cout << *pos << ' ';
// Insertamos el 4 otra vez y procesamos el valor
// que se devuelve.
pair<IntSet::iterator,bool> status = coll1.insert(4);
if (status.second) {
    cout << “Insertado.”
     << distance(coll1.begin(),status.first) + 1 << endl; }
else {  cout << "4 ya existe." << endl; }
```

Ejemplos uso del TDA Conjunto (set) (IV)

```
    // Creación de otro conjunto y asignación de los
    // elementos del primero de ellos.
    set<int> coll2(coll1.begin(),coll1.end());

    // Borrado de los elementos desde el principio hasta
    // el tres.
    coll2.erase (coll2.begin(), coll2.find(3));

    // Eliminación del elemento 5.
    int num = coll2.erase (5);
    cout << num << " elemento(s) eliminados." << endl;
}
```

Ejemplos uso del TDA Conjunto (set) (V)

```
// Creación de otro conjunto y asignación de los
// elementos del primero de ellos.
set<int> coll2(coll1.begin(),coll1.end());

// Borrado de los elementos desde el principio hasta
// el tres.
coll2.erase (coll2.begin(), coll2.find(3));

// Eliminación del elemento 5.
int num = coll2.erase (5);
cout << num << " elemento(s) eliminados." << endl;
}
```

Ejemplos uso del TDA Conjunto (set) (VI)

```
#include <iostream>
#include <set>
using namespace std;
int main ()
{
 set<int> c;
 c.insert(1); c.insert(2); c.insert(4); c.insert(5); c.insert(6);
 cout << "lower_bound(3): " << *c.lower_bound(3) << endl;
 cout << "upper_bound(3): " << *c.upper_bound(3) << endl;
 cout << "equal_range(3): " << *c.equal_range(3).first
     << " " << *c.equal_range(3).second << endl;
```

Ejemplos uso del TDA Conjunto (set) (VII)

```
cout << "lower_bound(5): " << *c.lower_bound(5) << endl;
cout << "upper_bound(5): " << *c.upper_bound(5) << endl;
cout << "equal_range(5): " << *c.equal_range(5).first
     << " "  << *c.equal_range(5).second << endl;
}
```

El TDA Bolsa (multiset)

Una bolsa es un caso especial de conjunto en el que se permite almacenar valores repetidos.

En la STL, el TDA Bolsa recibe el nombre de *multiset*.

Especificación del TDA Bolsa (multiset) (I)

/**

multiset<T,Comp>

TDA multiset::multiset, empty, clear, size, count, find, lower_bound, upper_bound, find, equal_range, insert, erase, begin, end, swap, ~multiset

Cada objeto del TDA Bolsa, modela un conjunto de elementos de la clase T.

Una bolsa es un contenedor que almacena elementos de manera ordenada que pueden estar repetidos.

Son objetos mutables. Residen en memoria dinámica.

*/

Especificación del TDA Bolsa (multiset) (II)

```
/**
   @brief Constructor primitivo.
   Crea una bolsa vacía.
 */
      multiset();
/**
   @brief Constructor de copia.
   @param b: bolsa que se copia.
   Crea una nueva bolsa que es copia de b.
*/
    multiset(const multiset<T> & b);
```

Especificación del TDA Bolsa (multiset) (III)

```
/**
  @brief Constructor de copia de un rango.
  @param inicio: apunta al elemento inicial a copiar.
  @param final: apunta al elmento final a copiar.
  Crea un nuevo conjunto que es copia del rango
      [inicio,final).
*/
multiset(iterator inicio, iterator final);
```

Especificación del TDA Bolsa (multiset) (IV)

```
/**
    @brief Informa si la bolsa está vacía.
    @return true, si la bolsa est vacía. false, en otro caso.
 */
 bool empty() const;
/**
   @brief Borra todos los elementos del contenedor.
   Deja el contenedor completamente vacío.
       Eficiencia: lineal.
*/
void clear();
```

Especificación del TDA Bolsa (multiset) (V)

```
/**
    @brief Obtiene el número de elementos contenidos.
    @return Número de elementos de la bolsa.
 */
     size_type size() const;
/**
    @brief Busca un elemento en la bolsa.
    @param clave: elemento a buscar.
    @return número de ocurrencias del elemento.
    comprueba si un elemento está contenido y devuelve cuántas veces. Eficiencia: lineal.
 */
     int count(const T & clave) const;
```

Especificación del TDA Bolsa (multiset) (VI)

```
/**
    @brief Busca un elemento en la bolsa.
    @param clave: elemento a buscar.
    @return Un iterador que apunta al primer elemento
            dentro de la bolsa que coincida con  la clave o
            end() en caso de  que no exista.
            Eficiencia: Logarítmica.
 */
    iterator find(const T & clave);
```

Especificación del TDA Bolsa (multiset) (VII)

```
/**
    @brief Determina la primera posición donde se
        insertará un elemento.
    @para clave: clave cuya ubicación se desea encontrar.
    @return Un iterador que señala a la primera posición
            donde una copia del argumento se insertaría
            de acuerdo con el criterio de ordenación.
    Según el critero de inserción que se use,
        determina la primera posición donde se
        insertaría una clave. Eficiencia: logartmica.
 */
    iterator lower_bound (const T & clave);
```

Especificación del TDA Bolsa (multiset) (VIII)

```
/**
    @brief Determina la primera posición donde se
           insertaría un elemento.
    @para clave: clave cuya ubicación se desea encontrar.
    @return Un iterador que señala a la primera posición
            donde una copia del argumento se insertaría
            de acuerdo con el criterio de ordenación.
    Según el critero de inserción que se use,
        determina la última posición donde se
        insertaría una clave. Eficiencia: logarítmica.
 */
iterator upper_bound (const T & clave);
```

Especificación del TDA Bolsa (multiset) (IX)

```
/**
   @brief Busca un elemento en la bolsa.
   @param clave: elemento a buscar.
   @return Un iterador constante que apunta al primer
           elemento dentro de la bolsa que
           coincida con la clave o end() en caso de
           que no exista.
           Eficiencia: logartmica.
*/
const_iterator find(const T & clave);
```

Especificación del TDA Bolsa (multiset) (X)

```
/**
   @brief Busca un elemento en la bolsa.
   @param clave: elemento a buscar.
   @return Un objeto de la clase pair, conteniendo un
           iterador  al primer elemento que coincide
           con la clave y otro al ltimo. En caso de
           que no exista los dos sern end().
           Eficiencia: logarítmica.
*/
pair<iterator,iterator> equal_range(const T & clave);
```

Especificación del TDA Bolsa (multiset) (XI)

```
/**
    @brief Inserta un elemento en la bolsa.
    @param clave: elemento a insertar.
    @return iterador dentro del multiset apuntando al
            elemento recién insertado.
    Inserta un elemento en la bolsa.
        Eficiencia: logarítmica.
 */
 iterator insert(const T & clave);
```

Especificación del TDA Bolsa (multiset) (XII)

```
/**
    @brief Borra todas las ocurrencias del argumento
           en la bolsa.
    @param clave: elemento a eliminar.
    @return Número de elementos eliminados.
    Al eliminarse el elemento, el tamaño se
        decrementa en tantos elementos como se
        hayan eliminado.
        Eficiencia: logarítmica.
 */
int erase(const T & clave);
```

Especificación del TDA Bolsa (multiset) (XIII)

```
/**
   @brief Borra elementos del rango [primero,ultimo).
   @param primero: primer elemento del rango.
          ultimo: último elemento del rango.
   @pre La bolsa no está vacía.
   Elimina los elementos incluidos en la rango
      pasado como argumento. El tamaño de la
      bolsa se decrementa según los elementos
      del rango.
*/
    void erase(iterator primero, iterator ultimo);
```

Especificación del TDA Bolsa (multiset) (XIV)

```
/**
   @brief Devuelve un iterador señalando al 1er  elem.
   @return Un iterador apuntando al primer elemento.
 */
 iterator begin();
/**
   @brief Devuelve un iterador constante al 1er elem.
   @return Un iterador constante apuntando al 1er
           elemento.
 */
 const_iterator begin() const;
```

Especificación del TDA Bolsa (multiset) (XIV)

```
/**
   @brief Devuelve un iterador sealando al último elem.
   @return Un iterador apuntando al último elemento.
*/
     iterator end();
/**
   @brief Devuelve un iterador constante señalando al
          último elemento de la bolsa.
   @return Un iterador constante apuntando al último
          elemento.
*/
     const_iterator end() const;
```

Especificación del TDA Bolsa (multiset) (XV)

```
/**
  @brief Intercambia el contenido del receptor y
         del argumento.
  @param c: bolsa a intercambiar con el receptor.
            ES MODIFICADO.
  Este método asigna el contenido del receptor al del
  parámetro y el del  parámetro al del receptor.
*/
void swap (multiset<T> & c);
```

Especificación del TDA Bolsa (multiset) (XVI)

```
/**
   @brief Destructor.
   @post El receptor es MODIFICADO.

   El receptor es destruido liberando todos los
   recursos que usaba. Eficiencia: lineal.
*/
     ~multiset();
```

Otras operaciones: ==, !=, <, >, <=, >= , =.

Ejemplos de uso del TDA Bolsa (multiset) (I)

```
#include <multiset>
using namespace std;

multiset<int> bolsaEnteros;
multiset<char, greater(char)> bolsaCaracteres;
multiset<long, less(long)> bolsaLong;

struct ltstr {
  bool operator()(const char* s1, const char* s2) const
  { return strcmp(s1, s2) < 0; }
};
```

Ejemplos de uso del TDA Bolsa (multiset) (II)

```
int main()
{
    typedef multiset<int,greater<int> > IntSet;
    IntSet coll1;
    coll1.insert(4);    coll1.insert(3);    coll1.insert(5);
    coll1.insert(1);    coll1.insert(6);    coll1.insert(2);
    coll1.insert(5);

    IntSet::iterator pos;
    for (pos = coll1.begin(); pos != coll1.end();++pos)
        cout << *pos << ' ';
    cout << endl;
```

Ejemplos de uso del TDA Bolsa (multiset) (III)

```
// Se inserta 4 de nuevo, procesando el valor devuelto
IntSet::iterator ipos = coll1.insert(4);
cout << "4 se inserta como elemento "
     << distance(coll1.begin(),ipos) + 1 << endl;

// Se crea otra bolsa por copia de rango.
multiset<int> coll2(coll1.begin(), coll1.end());

// Borrado de todos los elementos <= 3.
coll2.erase (coll2.begin(), coll2.find(3));
```

Ejemplos de uso del TDA Bolsa (multiset) (IV)

```
    // Se eliminan los elementos iguales a 5.
    int num;
    num = coll2.erase (5);
    cout << num << " elemento(s) eliminados." << endl;
}
```

El TDA Par (pair)

Este TDA es una utilidad que permite tratar un par de valores como una unidad.

Se utiliza en varios contenedores, en particular map y multimap, para gestionar sus elementos que son pares clave-valor.

Especificación del TDA Par (pair) (I)

```
/**
   @brief Constructor primitivo.
   Crea un par vacío.
 */
    pair();

/**
   @brief Constructor de copia.
   @param p: par que se copia.
   Crea un par que es copia de p.
*/
    pair(const pair<T1,T2> & p);
```

Especificación del TDA Par (pair) (II)

```
/**
   @brief Constructor a partir de dos valores.
   @param v1: primer valor.
   @param v2: segundo valor
   Crea un par a partir de los dos  valores.
*/
pair(const T1 & v1, const T2 & v2);
```

Operadores: =, ==, <

Variables Miembro: first, second.

Especificación del TDA Par (pair) (III)

Ejemplo de uso:

```
#include <pair>
using namespace std;

pair<int,String> miPar(3, ``Hola");

cout << miPar.first << `` ``  << miPar.second << endl;
```

El TDA Diccionario (map)

Un diccionario es un contenedor que almacena elementos (valores) identificados mediante una clave.

No existen pares (clave,valor) repetidos.

Anlogamente, al multiset para el set, existe el multimap para el map, permitiendo elementos con claves repetidas.

El TDA Diccionario (map) (II)

Ejemplos:

El diccionario de la RAE contiene parejas (palabra,definición).

El stock de un almacén: artículo, número de unidades que existe.

Un guía telefónica: abonado, número de teléfono.

Especificación del TDA Diccionario (map) (I)

/**

map<K,T,Comp>

TDA map::map, empty, clear, size, count, find, begin, end, equal_range, erase, insert, lower_bound, upper_bound, [], swap, ~map

Cada objeto del TDA Bolsa, modela un conjunto de elementos de la clase T.

Un diccionario es un contenedor que almacena elementos de manera ordenada, según sus correspondients claves.

Cada par (clave, valor) es único.

Son objetos mutables. Residen en memoria dinámica.

*/

Especificación del TDA Diccionario (map) (II)

```
/**
   @brief Constructor primitivo.
   Crea un diccionario vaco.
 */
    map();
/**
   @brief Constructor de copia.
   @param d: Diccionario que se copia.
   Crea un nuevo diccionario que es copia de d.
*/
    map(const map<key_type, T> & b);
```

Especificación del TDA Diccionario (map) (III)

```
/**
   @brief Constructor de copia de un rango.
   @param inicio: apunta al elemento inicial a copiar.
   @param final: apunta al elmento final a copiar.
   Crea un nuevo conjunto que es copia del rango [inicio,final).
*/
    map(iterator inicio, iterator final);
/**
   @brief Informa si el diccionario está vacío.
   @return true, si el diccionario está vacío; false, si no.
*/
    bool empty() const;
```

Especificación del TDA Diccionario (map) (IV)

```
/**
   @brief Borra todos los elementos del contenedor.
   Deja el contenedor completamente vacío.
       Eficiencia: lineal.
*/
      void clear();
 /**
  @brief Devuelve el número de elementos contenidos.
  @return Número de pares clave-valor del diccionario.
 */
      size_type size() const;
```

Especificación del TDA Diccionario (map) (V)

```
/**
    @brief Busca la clave en el diccionario.
    @param clave: identificador a buscar.
    @return Número de elementos con esa clave.
    Determina si hay algún elemento en el  diccionario con
     esa clave y  devuelve  el número de ellos.
        Eficiencia: lineal.
*/
int count(const key_type & clave) const;
```

Especificación del TDA Diccionario (map) (VI)

```
/**
   @brief Busca elementos con una clave dada.
   @param clave: identificador de los elementos.
   @return Un iterador al primer elemento dentro
                del diccionario que coincida con la
                clave o end() en caso contrario.
                Eficiencia: logartmica.
*/
iterator find(const key_type & clave);Ejemplos:
```

Especificación del TDA Diccionario (map) (VII)

```
/**
  @brief Busca elementos en el diccionario.
  @param clave: identificador de los elemento a buscar.
  @return Un objeto de la clase pair, conteniendo un
          iterador  al primer elemento que coincide
          con la clave y otro al último. En caso de
          que no exista los dos serán end().
          Eficiencia: logarítmica.
*/
pair<iterator,iterator> equal_range
                    (const key_range & clave);
```

Especificación del TDA Diccionario (map) (VIII)

```
/**
    @brief Borra el elemento del diccionario cuya
           clave coincida con la del argumento.
    @param clave: identificador del elemento a eliminar.
    @return 1 si el elemento estaba en el conjunto y
            ha sido eliminado, 0 en caso contrario.
    Al eliminarse el elemento, el tamaño se decrementa en
    una unidad.
    Eficiencia: logartmica.
 */
int erase(const key_type & clave);
```

Especificación del TDA Diccionario (map) (IX)

```
/**
    @brief Borra un elemento del diccionario.
    @param pos: señala el elemento a borrar.
    @pre El diccionario no está vacío.
    Elimina el elemento y el tamaño del diccionario se
    decrementa en una unidad.
 */
      void erase(iterator pos);
```

Especificación del TDA Diccionario (map) (X)

```
/**
    @brief Borra elementos del rango [primero,ultimo).
    @param primero: primer elemento del rango.
           ultimo: último elemento del rango.
    @pre El diccionario no está vacío.
    Elimina los elementos incluidos en el rango  pasado
    como argumento. El tamaño del diccionario se
    decrementa según los elementos del rango.
 */
void erase(iterator primero, iterator ultimo);
```

Especificación del TDA Diccionario (map) (XI)

```
/**
    @brief Inserta un elemento con su clave en el diccionario si
           no existe.
    @param par: Objeto de la clase pair conteniendo la
           clave y el elemento a insertar.
    @return Un objeto de la clase pair instanciada con un
           iterador y un valor booleano. Si la clave existe el iterador
           señala al elemento con dicha clave y false en el valor
           bool. Si no existe, el iterador señalará al elemento recién
           insertado, y true en la parte bool.
          Eficiencia: logarítmica.
*/
     pair<iterator, bool> insert(const value_type& Par);
```

Especificación del TDA Diccionario (map) (XII)

```
/**
    @brief Determina la primera posición donde se
           insertará un elemento.
    @para clave: clave cuya ubicación se desea encontrar.
    @return Un iterador que señala a la primera posición
            donde una copia del argumento se insertaría
            de acuerdo con el criterio de ordenación.
   Según el critero de inserción que se use,
   determina la primera posición donde se
   insertara una clave.    Eficiencia: logarítmica.
 */
iterator lower_bound (const value_type & clave);
```

Especificación del TDA Diccionario (map) (XIII)

```
/**
    @brief Determina la última posicin donde se
           insertaría un elemento.
    @para clave: clave cuya ubicación se desea encontrar.
    @return Un iterador que señala a la primera posición
            donde una copia del argumento se insertaría
            de acuerdo con el criterio de ordenación.
    Según el critero de inserción que se use,
    determina la última posición donde se
    insertaría una clave. Eficiencia: logarítmica.
 */
     iterator upper_bound (const value_type & clave);
```

Especificación del TDA Diccionario (map) (XIV)

```
/**
   @brief Inserta un elemento con clave k al diccionario
          en caso de que no exista. Si existe, sustituye
          el valor existente por el nuevo.
   @param k: Clave del elemento.
   @return El valor contenido en el diccionario cuya
           clave es la pasada como argumento.
           Eficiencia: logartímica.
*/
    T& operator[](const key_type& k);
```

Especificación del TDA Diccionario (map) (XV)

```
/**
   @brief Devuelve un iterador al 1er elemento del map.
   @return Un iterador apuntando al primer elemento.
*/
iterator begin();
/**
   @brief Devuelve un iterador constante al 1er elem.
   @return Un iterador constante apuntando al primer
           elemento.
*/
const_iterator begin() const;
```

Especificación del TDA Diccionario (map) (XVI)

```
/**
   @brief Devuelve un iterador al último elemento.
   @return Un iterador apuntando al último elemento.
*/
iterator end();

/**
   @brief Devuelve un iterador constante al último elem.
   @return Un iterador constante al último elemento.
*/
const_iterator end() const;
```

Especificación del TDA Diccionario (map) (XVII)

```
/**
 @brief Intercambia el contenido del receptor y
        del argumento.
 @param m: diccionario a intercambiar con el receptor.
           ES MODIFICADO.
 @doc Este mtodo asigna el contenido del
      receptor al del parmetro y el del
      parmetro al del receptor.
*/
void swap (map<K,T> & m);
```

Especificación del TDA Diccionario (map) (XVIII)

```
/**
    @brief Destructor.
    @post El receptor es MODIFICADO.

    El receptor es destruido liberando todos los
    recursos que usaba.
    Eficiencia: lineal.
 */
    ~map();
```

Otras operaciones: ==, !=, <, >, <=, >= , =

Ejemplos de uso del TDA Diccionario (map) (I)

```
#include <map>
using namespace std;

map<String,double> diccStringReal;
map<int,String> diccEntCad;
map<int,less(double)> diccEntReal;

struct ltstr {
  bool operator()(const char* s1, const char* s2) const
  { return strcmp(s1, s2) < 0;  } }
map<char *, int, ltstr> diccCharEnt;
```

Ejemplos de uso del TDA Diccionario (map) (II)

```
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{
    typedef map<string,float> StringFloatMap;
    StringFloatMap acciones;
```

Ejemplos de uso del TDA Diccionario (map) (III)

```
// Inserción de elementos.
  acciones["BASF"] = 369.50;   acciones["VW"] = 413.50;
 acciones["Daimler"] = 819.00;  acciones["BMW"]= 834.00;
 acciones["Siemens"] = 842.20;
// Impresión
   StringFloatMap::iterator pos;
   for (pos = acciones.begin(); pos != acciones.end();
        ++pos)
       cout << "acción: " << pos->first << "\t"
            << "precio: " << pos->second << endl;
   cout << endl;
```

Ejemplos de uso del TDA Diccionario (map) (IV)

```
    // Se doblan los precios de la acción
     for (pos = acciones.begin(); pos != acciones.end();
          ++pos)
         pos->second *= 2;

}
```

Ejemplos de uso del TDA Diccionario (map) (V)

Conversión de un número en su representación en letra:

934 = nueve tres cuatro

```
#include <iostream>
#include <string>
#include <map>
using namespace std;

typedef map<char, string, less<int> > dicc_CharString;

void construirMapa(dicc_CharString &);
void convertirCadena(string &, dicc_CharString &);
bool obtenerCadena(string &);
```

Ejemplos de uso del TDA Diccionario (map) (VI)

```
int main()
{
    dicc_CharString conversor;
    string cadena("");

    construirMapa(conversor);

    // Lectura de un número por la entrada estándar.
    while( obtenerCadena(cadena) )
      convertirCadena(cadena, conversor);

    return 0;
}
```

Ejemplos de uso del TDA Diccionario (map) (VII)

```
// Inicialización del conversor.
void construirMapa(dicc_CharString & dicc)
{
    dicc_CharString::iterator It;
    dicc insert(dicc_CharString::value_type('1',"uno"));
    dicc.insert(dicc_CharString::value_type('2',"dos"));
    //....
    dicc.insert(dicc_CharString::value_type('9',"nueve"));
    dicc.insert(dicc_CharString::value_type('.',"punto"));
    dicc.insert(dicc_CharString::value_type(',',"coma"));

    for(It = dicc.begin(); It != dicc.end(); ++It)
      cout << It->first << "," << It->second << endl;
}
```

Ejemplos de uso del TDA Diccionario (map) (VIII)

```
// Búsqueda de los dígitos de la cadena y conversión.
void convertirCadena(string &unaCadenaNumero,
                     dicc_CharString &dicc) {

  dicc_CharString::iterator unIterador;
  for(int index = 0; index < unaCadenaNumero.length();  index++)    {
     unIterador = dicc[unaCadenaNumero[index]];
     if( unIterador != dicc.end() )
       cout << unIterador->second << " " << flush;
     else   cout << "[err] " << flush;
   }
  cout << endl;
```

Ejemplos de uso del TDA Diccionario (map) (IX)

```
bool obtenerCadena(string &unaCadenaNumero)
{
  cout << "Introduce\"s\" para salir o introduce un número: ";
  cin >> unaCadenaNumero;
  return(unaCadenaNumero != "s");
}
```

El TDA Vector (vector)

Vector: secuencia de elementos que pueden ser accedidos mediante un índice, que indica la posición que ocupan en dicho contenedor.

El vector puede crecer o decrecer dinámicamente según las necesidades. También se pueden insertar elementos en cualquier posición o borrar ya existentes.

En la STL, el TDA Vector recibe el nombre de *vector*.

Especificación del TDA Vector (vector) (I)

/**

vector<T>

TDA vector::vector, reserve, size, capacity, [], push_back, begin, end, insert, erase, clear, at, swap, ~vector.

Cada objeto del TDA Vector, modela un vector de elementos de la clase T.

Secuencia dinámica de elementos identificados en ella por la posición que ocupan (índice). Su tamaño puede aumentarse o disminuirse de manera dinámica.

Son objetos mutables. Residen en memoria dinámica. */

Especificación del TDA Vector (vector) (II)

```
/**
  @brief Crea un vector dinámico vacío.
*/
    vector();
/**
  @brief Crea un vector dinámico.
  @param n Número de elementos. n>=0.
  Crea un vector de capacidad máxima n,
  con n valores almacenados todos nulos.
*/
    vector(int n=0);
```

Especificación del TDA Vector (vector) (III)

```
/**
  @brief Constructor de copia de un rango.
  @param inicio: apunta al elemento inicial a
    copiar.
  @param final: apunta al elmento final a copiar.
  Crea un nuevo conjunto que es copia del
  rango [inicio,final).
*/
 vector(iterator inicio, iterator final);
```

Especificación del TDA Vector (vector) (IV)

```
/**
  @brief Modifica la capacidad máxima.
  @param n Nueva capacidad máxima. n>=0
  Modifica el tamaño del vector de manera que
  ahora quepan en él n elemenos,  conservando
  los elementos que ya tuviera.
*/
     void reserve(int n);
```

Especificación del TDA Vector (vector) (V)

```
/**
  @brief Devuelve el número de elementos
  @return Número de elementos
*/
    int size() const;
/**
  @brief Devuelve la capacidad máxima del vector
  @return Número máximo de elementos
*/
    int capacity() const;
```

Especificación del TDA Vector (vector) (VI)

```
/**
  @brief Devuelve un elemento
  @param i Posición del elemento a devolver.
        0<=i<size()
  @return una referencia al elemento i-ésimo.
*/
    T& operator[](int i);
```

Especificación del TDA Vector (vector) (VII)

```
/**
  @brief Devuelve un elemento
  @param i Posición del elemento a devolver.
        0<=i<size()
  @return una referencia constante al elemento
         i-ésimo del vector
*/
const T& operator[](int i) const;
```

Especificación del TDA Vector (vector) (VIII)

```
/**
 @brief Añade un elemento al final del vector
 @param dato Elemento a añadir al final del vector
*/
    void push_back(const T &dato);
/**
  @brief Devuelve el inicio del vector
  @return Un iterador constante al inicio
*/
    const_iterator begin() const;
```

Especificación del TDA Vector (vector) (IX)

```
/**
  @brief Devuelve el final del vector
  @return Un iterador cte con valor igual al final
*/
const_iterator end() const;
/**
  @brief Devuelve el inicio del vector
  @return Un iterador al inicio
*/
iterator begin();
```

Especificación del TDA Vector (vector) (X)

```
/**
  @brief Devuelve el final del vector.
  @return Un iterador con valor igual al final.
*/
iterator end();
```

Especificación del TDA Vector (vector) (XI)

```
/**
  @brief Inserta un dato
  @param pos posición donde realizar la inserción
  @param x valor que se inserta
  @return Iterador indicando el dato insertado
*/
iterator insert(iterator pos, const T& x);
```

Especificación del TDA Vector (vector) (XII)

```
/**
  @brief Borra un dato
  @param pos posición que se borra
  @return Iterador indicando la posición siguiente
          a la borrada.  Eficiencia: lineal.
*/
iterator erase(iterator pos);
/**
  @brief Borra todos los elementos
  Deja el contenedor completamente vacío.
*/
 void clear();
```

Especificación del TDA Vector (vector) (XIII)

```
/**
  @brief Devuelve un elemento
  @param i Posición del elemento a devolver.
       0<=i<size()
  @return una referencia al elemento i-ésimo del
       vector
  Realiza comprobación de tipos si nos
  salimos fuera del rango.
*/
T & at (int i);
```

Especificación del TDA Vector (vector) (XIV)

```
 /**
   @brief Intercambia el receptor y el argumento.
   @param v: vector a intercambiar con el receptor.
            ES MODIFICADO.
*/
      void swap (vector<T> & v);
/**
  @brief Destructor.
  @post El receptor es MODIFICADO.
  Receptor destruido liberando recursos.
*/  ~vector();
```

Ejemplos de uso del TDA Vector (vector) (I)

```
#include <list>
using namespace std;

vector<int> vectorEnteros;
vector<String> vectorCadenas;
vector<MiTipo> vectorMiTipo;
vector<queue<double>> vectorDeColas;
```

Ejemplos de uso del TDA Vector (vector) (II)

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;
int main()
{
    vector<string> sentence;
    // Se reserva espacio para 5 elementos para
    // evitar que se tenga que resituar la memoria.
    sentence.reserve(5);
```

Ejemplos de uso del TDA Vector (vector) (III)

```
sentence.push_back("Hello,");
sentence.push_back("how");
sentence.push_back("are");
sentence.push_back("you");
sentence.push_back("?");

cout<< "max_size(): "<< sentence.max_size()<< endl;
cout<< "size():     "<< sentence.size()    << endl;
cout<< "capacity(): "<< sentence.capacity()<< endl;
// Intercambio de elementos.
swap (sentence[1], sentence[3]);
```

Ejemplos de uso del TDA Vector (vector) (IV)

```
    // Inserción de “always” antes de “?”
    sentence.insert(find(sentence.begin(),sentence.end(),
                    "?"), "always");

    // Se asigna "!" al último elemento.
    sentence.back() = "!";
}
```

El TDA Lista (list)

Intuitivamente:

Dado un conjunto D, una lista de elementos de dicho dominio es una secuencia finita y ordenada de elementos del mismo.

Formalmente:

Una lista es una aplicación de un conjunto de la forma:

$$l:1,2,..., n \rightarrow D$$

El TDA Lista (list) (II)

Una lista se suele representar de la forma:

$$l = < a_1, a_2, \ldots , a_n > \text{ con } a_i = a(i)$$

Nomenclatura de listas:

n: longitud de la lista

1, 2,..., n: posiciones de la lista

a_i: elemento que ocupa la posicin i-ésima

a_1: primer elemento de la lista

El TDA Lista (list) (III)

a_n: último elemento de la lista

n+1: posición final (tras el último elemento)

<>: lista vacía.

Orden inducido por la lista: a_i precede a a_{i+1} y sigue a a_{i-1}.

Ejemplos:

- La lista de alumnos matriculados en un curso.
- La lista de la compra.

Especificación del TDA Lista (list) (I)

/**

list<T>

TDA list::list, empty, clear, size, insert, erase, begin, end, push_back, push_front, pop_back, pop_front, back_front, ~list

Cada objeto del TDA lista de T, modela listas de objetos del dominio T. Cada objeto (instancia) del TDA lista tiene vinculado un conjunto de posiciones distinto.

Son mutables. Residen en memoria dinámica.

*/

Especificación del TDA Lista (list) (II)

```
/**
 @brief Constructor primitivo.
 Crea una lista vacía.
*/
      list();
/**
 @brief Constructor de copia.
 @param l: lista que se copia.
 Crea una lista que es copia de l.
*/
      list(const list<T> & l);
```

Especificación del TDA Lista (list) (III)

```
/**
  @brief Constructor de copia de un rango.
  @param inicio: apunta al elemento inicial a copiar.
  @param final: apunta al elmento final a copiar.
  Crea un nuevo conjunto que es copia del rango
  [inicio,final).
*/
list(iterator inicio, iterator final);
```

Especificación del TDA Lista (list) (IV)

```
/**
 @brief Informa si la lista está vacía.
 @return true, si la lista está vacía.  false, en otro caso.
*/
     bool empty() const;
/**
  @brief Borra todos los elementos del contenedor.
  Deja el contenedor completamente vacío.
      Eficiencia: lineal.
*/
     void clear ();
```

Especificación del TDA Lista (list) (V)

```
/**
   @brief Devuelve el número de elementos de la lista.
   @return número de elementos de la lista.
*/
      int size() const;
```

Especificación del TDA Lista (list) (VI)

```
/**
   @brief Inserta un elemento en la lista.
   @param p: posición delante de la que se inserta. Debe
        ser una posición válida para el objeto lista receptor.
   @param elemento: elemento que se inserta.
   @return posición del elemento insertado.
   Inserta un elemento con el valor 'elem' en la
   posición anterior a 'p'. MODIFICA al objeto receptor.
*/
      iterator insert(iterator p, const T & elemento);
```

Especificación del TDA Lista (list) (VII)

```
/**
  @brief Elimina un elemento de la lista.
  @param p: posición del elemento que se borra. Debe
            ser una posición válida para el objeto lista receptor.
  @return posición siguiente a la del elemento borrado.
  Destruye el objeto que ocupa la posición 'p' en el objeto
  lista receptor.
 */
    iterator erase(iterator p);
```

Especificación del TDA Lista (list) (VIII)

```
/**
   @brief Obtiene la primera posición de la lista.
   @return la primera posición de la lista receptora.
*/
      iterator begin();

/**
   @brief Obtener la primera posición de la lista.
   @return la primera posición de la lista receptora.
 */
      const_iterador begin() const;
```

Especificación del TDA Lista (list) (IX)

```
/**
   @brief Obtener la posición posterior al último elemento.
   @return la posición siguiente al último elemento.
*/
      iterator end() const;
/**
   @brief Obtener la posición posterior al último elemento.
   @return la posición siguiente al último elemento.
*/
      const_iterator end() const;
```

Especificación del TDA Lista (list) (X)

```
/**
  @brief Añade un elemento al final de la lista.
  @param dato Elemento a añadir al final de la lista.
*/
     void push_back(const T &dato);
/**
  @brief Añade un elemento al principio de la lista.
  @param dato Elemento a añadir al final de la lista.
*/
     void push_front(const T &dato);
```

Especificación del TDA Lista (list) (XI)

```
/**
  @brief Elimina el elemento del final de la lista.
*/
      void pop_back();
/**
  @brief Elimina el elemento del principio de la lista.
*/
      void pop_front();
/**
  @brief Devuelve el elemento al final de la lista.
*/
      T & back();
```

Especificación del TDA Lista (list) (XII)

```
/**
  @brief Devuelve el elmento del principio de la lista.
*/
     T & front();
 /**
 @brief Intercambia el contenido del receptor-argumento.
 @param l: lista a intercambiar con el receptor.
              ES MODIFICADO.
 Este método asigna el contenido del receptor al del
parámetro y el del parámetro al del receptor.
*/
       void swap (list<T> & l);
```

Especificación del TDA Lista (list) (XIII)

```
/**
  @brief Destructor.
  @post El receptor es MODIFICADO.

   El receptor es destruido liberando todos los
   recursos que usaba.
   Eficiencia: lineal.
*/
  ~list();
```

Otras operaciones: $==, !=, <, >, <=, >= , =$.

Ejemplos del uso del TDA Lista (list) (I)

```
#include <list>
using namespace std;
list<int> listaEnteros;
list<String> listaCadenas;
list<MiTipo> listaMiTipo;
list<list<double>> listaDelistas;
```

Ejemplos del uso del TDA Lista (list) (II)

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main()
{
    list<int> list1, list2;
    for (int i=0; i<6; ++i) {
        list1.push_back(i);
        list2.push_front(i);
    }
```

Ejemplos del uso del TDA Lista (list) (III)

```
    list<int>:const_iterator iter;

    for (iter=list1.begin(); iter!= list1.end(); iter++)
        list2.push_back(*iter);

}
```

Ejemplos del uso del TDA Lista (list) (IV)

```
void Ordenar( list<int> & l)
  /**
       @param l es la lista a ordenar
       @brief Devuelve la Lista l ordenada
  */
{
  list<int>::iterator itr,ant;
  itr = l.begin();   ++itr;
  while (itr != l.end()){
    ant = itr;
    --ant;
```

Ejemplos del uso del TDA Lista (list) (IV)

```
    if( *ant > *itr){
      for ( ; *ant>*itr && ant!=l.begin(); --ant);

      if (ant==l.begin() && (*ant>*itr))
        l.insert(l.begin(),*itr);
      else l.insert(++ant,*itr);
      itr = l.erase(itr);
    }
    else ++itr;


  } // While
} // función
```

Ejemplos del uso del TDA Lista (list) (IV)

```
void InsertaOrdenado(int x, list<int> &l)
  /**
     @param  x es el elemento a insertar
     @param  l es una Lista ordenada
     @brief Inserta el elemento x en l, ordenadamente
  */
{
  list<int>::iterator i=l.end();  --i;
  if ( *i<x )  l.insert(l.end(),x);
  else {
    for (--i; *i>x && i!=l.begin();--i);
    if ( *i>x && i==l.begin()) l.insert(l.begin(),x);
         else l.insert(++i,x);
  }}
```

Ejemplos del uso del TDA Lista (list) (IV)

```
template <class T>
void EliminaDuplicados(list<T> &  l)
/**
   @brief Elimina los elementos duplicados de una lista.
   @param: lista a procesar. Es MODIFICADO.
   */
{
  for (list<T>::iterator p = l.begin();  p != l.end(); ++p)
    {
      list<T>::iterator q = p;    ++q;
      while (q != l.end())   {
          if (*p == *q)   q = l.erase(q);
          else  ++q;
        }
    }}
```

Ejemplos del uso del TDA Lista (list) (IV)

```
template <class T>
void comunes (const list<T> & l1, const list<T> & l2, list<T> & lsal )
{
  list<T>::const_iterator i1,i2;
  bool enc = false;
  for (i1= l1.begin(); i1!=l1.end();++i1){
    enc= false;
    for (i2 = l2.begin(); i2!= l2.end() && !enc ; ++i2)
      if (*i1 == *i2) {
          enc = true;
          lsal.insert( lsal.end(), *i2);  }
  } // bucle for
} // función comunes.
```

Ejemplos del uso del TDA Lista (list) (IV)

```
template<class T>
class par {
  public:  T elem;     int contador;  };


template <class T>
void crecimiento (const list<T> & l1, list<par<T> > & lc)
  {
/**
    @brief lc es una lista donde, para cada elemento de  l1 contiene el numero de elementos consecutivos  en l1 que, precediendo al elemento, verifican  que son menores que él.
*/
```

Ejemplos del uso del TDA Lista (list) (IV)

```
list<T>::const_iterator it, r;
par<T> aux;    aux.contador = 0;   aux.elem =*(l1.begin());
lc.insert(lc.begin(),aux);
it = l1.begin();    ++it;
for ( ; it!= l1.end(); ++it){
  aux.contador = 0;  aux.elem = *it;
  r = it;   --r;
  while (r!= l1.begin() && *r < *it){
    --r;
    aux.contador++;
  } // while
  if (r== l1.begin() && *r < *it)  aux.contador++;
  lc.insert(lc.end(),aux);
}// for } // función
```

Un comentario final

Observando las diferentes operaciones de los contenedores set, multiset, map, multmap,vector y list, podemos concluir que existe una serie de operaciones comunes a todos ellos:

- Construtor primitivo.
- Constructor de copia.
- Constructor de copia a partir de un rango.
- Destructor.

Un comentario final (II)

empty.

Operadores: ==, !=, <, >, <=, >=, =.

swap.

Begin y end.

rbegin (devuelve reverse_iterator y const_reverse_iterator).

rend (devuelve reverse_iterator y const_reverse_iterator).

clear.

Un comentario final (III)

A cada contenedor se le añaden las características propias de su funcionalidad.

Esto nos permitirá, por ejemplo, realizar sobre ellos operaciones independientemente del tipo de contenedor, como por ejemplo, el recorrido de los mismos, o la búsqueda de un elemento mediante la función find, entre otras muchas.