TDA Conjunto Mutaciones

V5

Generado por Doxygen 1.8.9.1

Viernes, 2 de Diciembre de 2016 00:24:44

Índice

1. Iterando sobre el conjunto

Versión

v5 Iteradores

Autor

Juan F. Huete y Carlos Cano

1.1. Introducción

En la práctica anterior hemos construido el conjunto genérico (se puede particularizar sobre distintos tipos de datos y distintos criterios de comparación). Aunque estaba dotado de un mecanismo para poder iterar sobre las entradas, este mecanismo lo heredaba directamente de la clase vector subyacente, pudiendo provocar errores en la representación (no hay ocultamiento de información).

Por ejemplo, sería posible hacer

```
conjunto<mutacion,less<mutacion> > X;
mutacion aux1,aux2;
aux1.setPos(1234);
aux1.setChr("MT");
conjunto<mutacion,less<mutacion> >::iterator it;
it = X.find(aux1);
if (it!=X.end())
  {aux2.setPos(4321);
```

```
   aux2.setChr("1");
   *it = aux2;  //VIOLAMOS EL INVARIANTE DE LA REPRESENTACION
}
```

En este práctica lo que proponemos es dotar a este conjunto de mecanismos de iteración específicos, siguiendo el estándar que se considera para acceder a los elementos de un contenedor, sin necesidad de conocer las particularidades internas de la implementación.

1.2. Generar la Documentación.

Al igual que en la práctica anterior la documentación se entrega mediante un fichero documentacion.pdf, así como mediante un fichero zip que contiene todos los fuentes junto a los archivos necesarios para generar la documentación (en latex y html). Para generar los ficheros html con la documentación de la misma es suficiente con ejecutar desde la línea de comandos

```
doxygen doxPractica.txt
```

Como resultado nos genera dos directorios, uno con la documentación en html y el otro con la documentación en latex.

Se entregan los ficheros de especificación nueva para el TDA conjunto. Estos ficheros incluyen algunas modificaciones que viene dadas por el uso de los iteradores.

- conjunto.h En el nuevo fichero conjunto.h se entrega la nueva especificación de la clase conjunto, donde además se le añade la especificación de dos iteradores. Se os pide implementar los distintos métodos así como el código necesario para demostrar el correcto funcionamiento del mismo.

Pasamos a detallar cada una de las partes de la práctica.

1.3. Iteradores sobre conjunto.

Casi todos los contenedores disponen de una (o varias) clases asociada llamada iterator. Para poder asociar el iterador al contenedor una alternativa es añadir una clase anidada (una clase que se define dentro de la clase contenedora). Ambas clases están estrechamente relacionadas, por lo que es muy usual que se desee que tanto el contenedor como el iterador sean clases amigas. Así, cuando se crea una clase friend anidada es conveniente declarar primero el nombre de la clase y después definir la clase. Así evitamos que se confunda el compilador.

```
template <typename T, class CMP>
class conjunto{
public:
        typedef T value_type;
    typedef unsigned int size_type;

    class iterator;
    class const_iterator;
    class impar_iterator;  // Itera sobre mutaciones con posición impar
    class secure_iterator; // itera asegurando que las posiciones son correctas dentro del vector.
    class const_impar_iterator;  // Itera sobre mutaciones con posición impar
    class const_secure_iterator; // itera asegurando que las posiciones son correctas dentro del vector.
   ....

   impar_iterator ibegin();
   const_impar_iterator cibegin( );
   secure_iterator sbegin( );
   const_secure_iterator csbegin( );
   impar_iterator iend();
   const_impar_iterator ciend( );
   secure_iterator send( );
   const_secure_iterator csend( );


   ....
   class impar_iterator {
      //definicion del iterator
      public:
    impar_iterator();
```

```
        ....
      private:
         friend class conjunto<T,CMP>; // declaramos conjunto como amigo de la clase
        ....

    }; // end de la clase iterator

   private:
        friend class impar_iterator; // declaramos el iterador como amigo de la clase

}; // end de la clase conjunto
```

Es importante notar que el tipo asociado al iterador es conjunto<T,CMP>::xxx_iterator Por tanto, para declarar un conjunto y un iterator sobre dicho conjunto debemos hacer

```
conjunto<mutacion,less<mutacion> > C;
conjunto<mutacion,less<mutacion> >::iterator it;
conjunto<mutacion,less<mutacion> >::impar_iterator iit;

for (it = C.begin() ; it!=C.end();++it) //Itera sobre todos los elementos del conjunto.
  cout « *it « endl;

 //iit Itera sobre todos las mutaciones en posiciones impares
for (iit = C.ibegin(); iit!= C.iend();++iit)
  cout « *iit « endl;
```

1.4. begin y end

Para poder iterar sobre los elementos del contenedor, debemos dotarlo de nuevos métodos (que siguiendo en estándar de la Standard Template Library llamaremos begin y end). En sus distintos formatos begin devuelve un iterator que apunta al primer elemento del contenedor (primer elemento que satisface las condiciones por las que se itera), mientras que end (en sus distintas versiones) por su parte nos devuelve un iterator que apunta «al final» del contenedor. Es importante recordar que la posición final del contenedor no es una posición válida del mismo, esto es, no hay ningún elemento en dicha posición (es conveniente pensar que es la posición siguiente al último elemento del contenedor). Por ello, no es correcto dereferenciar el elemento alojado en dicha posición (∗end()).

Además podemos ver el uso de paréntesis para acceder a los elementos (∗it).getID(). En este caso, si hacemos ∗it.getID, dada la precedencia de los operadores, primero se evaluaría el operador "."

```
(*it).getID() // Correcto
*it.getID() // Incorrecto, primero evalúa it.getID()
```

Además del begin y end que devuelven el iterator, y siguiendo la filosofía del estándar C++11, implementaremos dos métodos, el cbegin y el cend que devuelven los const_iterator

```
conjunto<mutacion,less<mutacion> >::const_iterator csit = C.cbegin();
```

1.4.1. impar_iterator

En esta práctica debemos destacar el comportamiento de impar_iterator. Dicho iterador nos permitirá iterar sobre todos los elementos que contengan una determinada mutación que está en posiciones impares del cromosoma. Obviamente, este iterador sólo será valido cuando el tipo elemento sobre el que se particulariza el conjunto tenga definido el método getPos() que devuelva un entero. Por tanto, valdría para un conjunto<mutacion,less<mutacion> > pero no para un conjunto<enfermedad,less<enfermedad> >.

```
@brief devolver primera posición del elemento que se encuentra en posiciones impares
@return un iterador que apunta a la primera mutación con posición impar dentro del conjunto,  si no hay
      devuelve iend
conjunto<T;CMP>::impar_iterator conjunto<T,CMP>::ibegin( );

@brief devolver final posiciones impares
@return nos debe devolver un iterador que apunta a la posición final del mismo.
conjunto<T;CMP>::impar_iterator conjunto<T,CMP>::iend( );
```

El método iend() puede puede coincidir con el end() del vector<mutaciones>

1.4.2. impar_iterator

Con respecto al secure_iterator, nos permitirá asegurarnos que siempre el iterador apunta a una posición valida del vector, en caso contrario debemos de abortar el programa. Para ello utilizaremos el método assert (de la biblioteca assert.h). Este chequeo se deberá hacer en todos los métodos que puedan provocar un error, como operator∗, operator++, operator--, etc.

```
#include <assert.h>
....
class secure_iterator{
....
}

....
const T & xxx::secure_iterator::operator*(){
     ....
     assert (posicion correcta); // entre el begin y end del vector al que apunta
     return elemento_en_posicion;
}
```

1.5. Representación del iterator

Un iterator de la clase conjunto nos debe permitir el acceso a los datos almacenados en el conjunto propiamente dicho. Una primera alternativa sería representar el iterator como un iterador sobre el vector, directamente como lo hemos considerado en las prácticas anteriores,

```
class conjunto{
   ....
  typedef  vector<value_type>::iterator iterator;
   ...
```

o implementando todo el iterador como podría ser

```
class conjunto{
....
  class iterator {
    ....
     value_type & operator*(); // NO seria correcto
    ......
    private:
      vector<value_type>::iterator it_v; // Puntero a la entrada del vector.
   };
};
```

Sin embargo, con ambas representaciones sería posible violar el invariante de la representación del TDA conjunto. Así, el usuario de la clase podría modificar el contenido de la clave ejecutando

```
conjunto<mutacion,less<mutacion> > X;
mutacion aux1,aux2;
aux1.setPos(1234);
aux1.setChr("MT");
conjunto<mutacion,less<mutacion> >::iterator it

*(X.begin()) = aux1; //VIOLAMOS INVARIANTE

X.find(aux1);
if (it!=X.end())
  {aux2.setPos(4321);
   aux2.setChr("1");
   *it = aux2;  //VIOLAMOS EL INVARIANTE DE LA REPRESENTACION
}
```

Esto nos daría problemas pues estaríamos modificando la clase, y particularmente al asumir los datos ordenados, el conjunto podría dejar de estar ordenado, no cumpliría el invariante de la representación. A partir de este momento las operaciones de búsqueda e inserción dejarían de funcionar correctamente.

Para solucionar el problema es necesario que todos los iteradores del conjunto devuelvan una referencia constante a los elementos almacenados en el mismo

```
class conjunto{

....
  class iterator {
     ....
    const value_type & operator*();
    ......
    private:
    ....
   };
....
  class const_iterator {
     .....
     const value_type & operator*();
     ....
     private:
     ....
};
```

1.6. SE PIDE

En concreto se pide implementar los métodos asociados a los iteradores de la clase conjunto.

En este caso, para realizar la práctica, el alumno deberá modificar los ficheros de implementación (.hxx).

De igual forma se debe modificar el fichero prueba.cpp de manera que se demuestre el correcto comportamiento del conjunto cuando se instancia con distintos tipos. Debe modificar el fichero para añadir más ejemplos donde se demuestre el uso de iterators y const_iterators de forma correcta. A modo ilustrativo se entrega el fichero principal.cpp.

1.6.1. A ENTREGAR

El alumno debe entregar los siguientes ficheros, con las correcciones necesarias para poder trabajar

- documentacion.pdf
- ficheros.zip

Dicha entrega tiene como límite el Domingo 4 de Diciembre.

2. Indice de archivos

2.1. Lista de archivos

Lista de todos los archivos con descripciones breves:

3. Documentación de archivos

3.1. Referencia del Archivo documentacion.dox

3.2. Referencia del Archivo principal.cpp

```
#include "mutacion.h"
#include <fstream>
#include "conjuntoIt.h"
```

'typedefs'

- typedef conjunto< int, int > aaa

Funciones

- template<class CMP >
  bool load (conjunto< mutacion, CMP > &cm, const string &s)

  lee un fichero de mutaciones, linea a linea
- template<typename T >
  int distancia (T ita, T itb)
- int main (int argc, char *argv[ ])

3.2.1. Documentación de los 'typedefs'

3.2.1.1. typedef conjunto<int,int> aaa

3.2.2. Documentación de las funciones

3.2.2.1. template<typename T > int distancia (T *ita,* T *itb*)

3.2.2.2. template<class CMP > bool load (conjunto< mutacion, CMP > & *cm,* const string & *s*)

lee un fichero de mutaciones, linea a linea

Parámetros

in	*s*	nombre del fichero
in,out	*cm*	objeto tipo conjunto sobre el que se almacenan las mutaciones

Devuelve

true si la lectura ha sido correcta, false en caso contrario

3.2.2.3. int main (int *argc,* char * *argv[ ]*)

Índice alfabético