TDA Conjunto Genérico

V2

Generado por Doxygen 1.8.11

Índice

1. PRACTICA TEMPLATE

Versión

v2

Autor

Juan F. Huete y Carlos Cano.

1.1. Introducción

En la práctica anterior se os pidió la implementación del tipo conjunto de mutaciones. En esta práctica el objetivo es seguir avanzando en el uso de las estructuras de datos, particularmente mediante el uso de plantillas (templates) para la definición de tipos de datos genéricos.

Nuestro objetivo es dotar al TDA conjunto de la capacidad de almacenar elementos de cualquier tipo. Teniendo en cuenta que los elementos se almacenan de forma ordenada, es necesario que el tipo de dato con el que se instancia el conjunto tenga definido una relación de preorden total, R, esto es:

- Para todo x,y: xRy || yRx
- Para todo x,y,z: Si xRy && yRz entonces xRz

Por tanto R es una relación binaria que toma como entrada dos elementos del mismo tipo y como salida nos devuelve un booleano. Ejemplos de este tipo de relaciones son el operator< (o el operator>), ya definidos sobre la clase mutación y enfermedad

Tanto el tipo de dato con el que particularizar el conjunto como el criterio de ordenación serán proporcionados a la hora de definir un conjunto.

```
template <typename T, typename CMP> class conjunto;
```

en la plantilla T hace referencia al tipo de dato, y CMP hace referencia al criterio de comparación interno (functor).

En la clase se declarará un objeto tipo CMP, que llamaremos comp. Así, la expresion comp(a,b) devuelve true si se considera que a precede b en la relación de preorden. Esta relación será utilizada por el "conjunto" para:

- decidir cuando un elemento precede a otro en el contenedor,
- pero también a la hora de determinar cuando dos elementos son equivalentes: para determinar cuando dos elementos serán considerados iguales con respecto a la relacion tendremos en cuenta que

```
Si (!comp(a,b) && !comp(b,a))  entonces necesariamente a==b.
```

1.2. Uso de templates

Hasta ahora, en un conjunto de mutaciones los elementos se encuentran almacenadas en orden siguiendo el criterio de comparación dado por el operador< (primero se compara cromosoma y en caso de empate se compara la posición). Sin embargo nos podríamos plantear otros conjuntos de elementos. Así, podríamos tener

```
#include "conjunto.h"
...
// declaracion de tipos básicos:

conjunto<mutacion,less<mutacion > > Xl;  // elementos ordenados en orden creciende  (operator< sobre
       mutacion)
conjunto<mutacion,greater<mutacion > > Xg;  // elementos ordenados en
       orden creciende  (operator> sobre mutacion)

conjunto<enfermedad,greater<enfermedad> > Yg; // elementos ordenados
       en orden decreciente (operator> sobre enfermedad)

// declaración de tipos más complejos:

conjunto<mutacion,less<mutacion> >::iterator itl;
conjunto<enfermedad,greater<enfermedad> >
      ::iterator itg;


...
```

Hay que notar que en este ejemplo Xl y Xg representan a tipos distintos, esto es un conjunto ordenado en orden creciente NO SERÁ del mismo tipo que un conjunto ordenado en orden decreciente. De igual forma, itl e itg tampoco serán variables del mismo tipo, por lo que no podríamos hacer entre otras cosas asignaciones como

```
Xl=Yg; // ERROR  un conjunto ordenado en orden descendente no puede ser asignado a un conjunto ordenado en
       orden descendente
...
itg=itl; // igual que en el caso anterior, apuntan a elementos distintos
```

En este caso, para realizar la práctica, el alumno deberá modificar tanto el fichero de especificación, conjunto.h, de forma que la propia especificación indique que trabaja con parámetros plantilla, como los ficheros de implementación (.hxx) de la clase conjunto.

De igual forma se debe modificar el fichero principal.cpp de manera que se demuestre el correcto comportamiento del conjunto cuando se instancia bajo distintos tipos de datos y distintos criterios de ordenación, en concreto debemos asegurarnos que utilizamos los siguientes criterios de ordenación:

- conjunto de mutacion creciente por cromosoma/posicion
- conjunto de mutacion decreciente por cromosoma/posicion
- conjunto de mutacion creciente por id
- conjunto de mutacion decreciente por id
- conjunto de enfermedad por orden creciente
- conjunto de enfermedad por orden decreciente

Para los dos primeros casos, y teniendo en cuenta que tenemos sobrecargado los operadores relacionales para mutación, es suficiente con utilizar las clases genéricas less<T> y greater<T> definidas en functional (#include <functional>). Sin embargo, para el resto de casos del conjunto de mutaciones debemos implementar los functores que nos permitan realizar la correcta comparación entre mutaciones.

1.3. functor

Para realizar dichas comparaciones utilizaremos una herramienta potente de C++: un functor (objeto función). Un functor es una clase en C++ que actua como una función. Un functor puede ser llamado con una sintaxis familiar a la de las funciones en C++, pudiendo devolver valores y aceptar parámetros como una función normal.

Por ejemplo, si queremos crear un functor que compare dos mutaciones teniendo en cuenta el orden de ID, podríamos hacer

```
mutacion x,y;
...
ComparacionPorID miFunctor;
cout « miFunctor(x,y) « endl;
```

Aunque miFunctor es un objeto, en la llamada miFunctor(x,y) la tratamos como si estuviésemos invocando a una función tomando x e y como parámetros.

Para crear dicho functor, creamos un objeto que sobrecarga el operador() como sigue

```
class ComparacionPorID {
 public:
   bool operator()(const mutacion &a, const mutacion &b) {
     return (a.getID() < b.getID()); // devuelve verdadero si el ID de a precede al ID de b
 }
};
```

1.4. Generalizando el conjunto.

Para poder extender nuestro conjunto hemos de dotarlo de la capacidad de poder definir el criterio de ordenación. Para ello vamos a considerar un caso simplificado (que no se corresponde exactamente con lo que se pide en la práctica) donde ilustraremos su uso

```
template <typename T, typename CMP>
class conjunto {
  public:
    ....
    pair<typename conjunto<T,CMP>::iterator,bool> insert( const T & c);

  private:
    vector<T> vdatos; //donde se almacenan los datos
    CMP comp;
};
```

Como hemos dicho, el nombre del tipo ahora es conjunto<T,CMP> y no conjunto. Distintas particularizaciones dan lugar a tipos también distintos. Ahora, en el fichero conjunto.hxx debemos de implementar cada uno de los métodos, recordemos que cada uno de ellos pertenece a la clase conjunto<T,CMP> y por tanto se implementa considerando

```
tipoRetorno conjunto<T,CMP>::nombreMetodo( parametros ...)
```

Pasamos a ver la implementación de los métodos:

1.4.1. Insert

El método insert asume como prerequisito que el conjunto está ordenado según el criterio dado por CMP, y por tanto debe asegurar que tras insertar una nueva entrada dicho conjunto siga ordenado. Por ejemplo, podríamos hacer algo del tipo:

```
pair<typename conjunto<T,CMP>::iterator,bool>
                              conjunto<T,CMP>::insert( const T & c){

      pair<typename conjunto<T,CMP>::iterator,bool> salida;

      bool fin = false;
      for (auto it = vdatos.begin(); it!=vdatos.end() && !fin; ){
        if (comp(*it,c) ) it++;
        else if (!comp(*it,c) && !comp(c,*it)){ // equivalentes segun CMP
            salida.first = vdatos.end();
            salida.second = false;
            fin = true;
        }else {
            salida.first = vdatos.insert(it,c);
            salida.second = fin = true;
        }
      } // del for
      if (!fin){
        salida.first = vdatos.insert(vdatos.end(),c);
        salida.second = true;
      }
      return salida;
}
```

En este caso comp(∗it,c) hace referencia a una comparación genérica entre elementos de tipo T definida por la relación de orden con la que se haya particularizado el conjunto. Así si hemos definido

```
conjunto<mutacion,ComparacionPorID> cID;
```

en este caso comp es un objeto de la clase ComparacionPorID, y mediante la llamada comp(.,.) lo que estamos haciendo es llamar a la "función" que me compara dos mutaciones teniendo en cuenta su ID.

Finalmente, debemos tener cuidado a la hora de realizar comparaciones y la semántica de las mismas, como se muestra en el caso en que comparamos cuando dos elementos son "iguales". Así, si en lugar de realizar la comparación

```
if (!comp(*it,c) && !comp(c,*it)){ // equivalentes segun CMP
}
```

hubiésemos utilizado

```
If (*it==c) { //igualdad
```

estaríamos haciendo la llamada a la comparación de igualdad entre mutaciones (definida mediante la comparación de cromosoma/posicion) por lo que podría funcionar correctamente el método cuando particularizamos con

```
Conjunto<mutacion,less<mutacion> > X;
```

Sin embargo, si el conjunto está definido como conjunto<mutacion,ComparacionPorID>, utilizar el mismo código para realizar la búsqueda no funcionaría correctamente: los elementos están ordenados en orden creciente de ID y la comparación de igualdad se hace por cromosoma/mutacion.

1.4.2. SE PIDE

Por tanto, se pide la implementación de los métodos de la clase conjunto genérico y su prueba de funcionamiento correcto en los supuestos planteados anteriormente

- conjunto de mutacion creciente por cromosoma/posicion
- conjunto de mutacion decreciente por cromosoma/posicion
- conjunto de mutacion creciente por id
- conjunto de mutacion decreciente por id
- conjunto de enfermedad por orden creciente
- conjunto de enfermedad por orden decreciente

Dicha entrega se debe realizar antes del 18 de Noviembre a las 23:59 horas.

2. Lista de tareas pendientes

Clase conjunto< T, CMP >
Implementa esta clase siguiendo la especificación asociada

3. Índice de clases

3.1. Lista de clases

Lista de las clases, estructuras, uniones e interfaces con una breve descripción:

4. Indice de archivos

4.1. Lista de archivos

Lista de todos los archivos con descripciones breves:

5. Documentación de las clases

5.1. Referencia de la plantilla de la Clase conjunto< T, CMP >

Clase conjunto.

```
#include <conjunto.h>
```

Tipos públicos

- typedef T value_type
- typedef unsigned int size_type
- typedef vector< value_type >::iterator iterator
- typedef vector< value_type >::const_iterator const_iterator

Métodos públicos

- conjunto ()

  constructor primitivo.
- conjunto (const conjunto< T, CMP > &d)

  constructor de copia
- iterator find (const value_type &s)

  busca una entrada en el conjunto
- const_iterator find (const value_type &s) const
- size_type count (const value_type &e) const

  cuenta cuantas entradas coinciden con los parámetros dados.
- pair< iterator, bool > insert (const value_type &val)

  Inserta una entrada en el conjunto.
- pair< iterator, bool > insert (value_type &val)
- iterator erase (const iterator position)

  Borra una entrada en el conjunto . Busca la entrada y si la encuentra la borra.
- size_type erase (const value_type &val)
- void clear ()

  Borra todas las entradas del conjunto, dejandolo vacio.
- size_type size () const

  numero de entradas en el conjunto
- bool empty () const

  Chequea si el conjunto esta vacio (size()==0)
- conjunto & operator= (const conjunto &org)

  operador de asignación
- conjunto::iterator begin ()

  begin del conjunto
- conjunto::iterator end ()

  end del conjunto
- conjunto::const_iterator cbegin () const

  begin del conjunto
- conjunto::const_iterator cend () const

  end del conjunto
- iterator lower_bound (const value_type &val)

  busca primer elemento que no está por debajo ('antes', '<') de los parámetros dados.
- const_iterator lower_bound (const value_type &val) const
- iterator upper_bound (const value_type &val)

  busca primer elemento por encima ('después', '>') de los parámetros dados.
- const_iterator upper_bound (const value_type &val) const

Métodos privados

- bool cheq_rep () const

  Chequea el Invariante de la representacion.

Atributos privados

- vector< value_type > vm
- CMP comp

5.1.1. Descripción detallada

template<typename T, typename CMP>
class conjunto< T, CMP >

Clase conjunto.

conjunto<T,CMP>::conjunto, find, size, insert, erase,

Tipos conjunto<T,CMP>::value_type, conjunto<T,CMP>::size_type, conjunto<T,CMP>::iterator, conjunto<T,↩ CMP>::const_iterator

Descripción

Un conjunto es un contenedor que permite almacenar en orden creciente un conjunto de elementos no repetidos. En nuestro caso el conjunto va a tener un subconjunto restringido de métodos (inserción de elementos, consulta de un elemento, etc). Este conjunto "simulará" un conjunto de la stl, con algunas claras diferencias pues, entre otros, no estará dotado de la capacidad de iterar (recorrer) a través de sus elementos.

Asociado al conjunto, tendremos el tipo

```
conjunto::value_type
```

que permite hacer referencia al elemento almacenados en cada una de las posiciones del conjunto, en nuestro caso mutaciones (SNPs) o enfermedades. Es requisito que el tipo conjunto::value_type tenga definida una relación de orden, CMP, y el operador de asignación, operator= .

El número de elementos en el conjunto puede variar dinámicamente; la gestión de la memoria es automática.

Tareas pendientes Implementa esta clase siguiendo la especificación asociada

5.1.2. Documentación de los 'Typedef' miembros de la clase

5.1.2.1. template<typename T, typename CMP> typedef vector<value_type>::const_iterator conjunto< T, CMP >::const_iterator

5.1.2.2. template<typename T, typename CMP> typedef vector<value_type>::iterator conjunto< T, CMP >::iterator

5.1.2.3. template<typename T, typename CMP> typedef unsigned int conjunto< T, CMP >::size_type

5.1.2.4. template<typename T, typename CMP> typedef T conjunto< T, CMP >::value_type

5.1.3. Documentación del constructor y destructor

5.1.3.1. template<typename T, typename CMP> conjunto< T, CMP >::conjunto ( )

constructor primitivo.

5.1.3.2. template<typename T, typename CMP> conjunto< T, CMP >::conjunto (const conjunto< T, CMP > & *d*)

constructor de copia

Parámetros

in	*d*	conjunto a copiar

5.1.4. Documentación de las funciones miembro

5.1.4.1. template<typename T, typename CMP> conjunto::iterator conjunto< T, CMP >::begin ( )

begin del conjunto

Devuelve

Devuelve un iterador al primer elemento del conjunto. Si no existe devuelve end

Postcondición

no modifica el conjunto.

5.1.4.2. template<typename T, typename CMP> conjunto::const_iterator conjunto< T, CMP >::cbegin ( ) const

begin del conjunto

Devuelve

Devuelve un iterador constante al primer elemento del conjunto. Si no existe devuelve cend

Postcondición

no modifica el conjunto.

5.1.4.3. template<typename T, typename CMP> conjunto::const_iterator conjunto< T, CMP >::cend ( ) const

end del conjunto

Devuelve

Devuelve un iterador constante al final del conjunto (posicion siguiente al ultimo.

Postcondición

no modifica el conjunto.

5.1.4.4. template<typename T, typename CMP> bool conjunto< T, CMP >::cheq_rep ( ) const [private]

Chequea el Invariante de la representacion.

Devuelve

true si el invariante es correcto, falso en caso contrario

5.1.4.5. template<typename T, typename CMP> void conjunto< T, CMP >::clear ( )

Borra todas las entradas del conjunto, dejandolo vacio.

Postcondición

El conjunto se modifica, quedando vacio.

5.1.4.6. template<typename T, typename CMP> size_type conjunto< T, CMP >::count (const value_type & *e*) const

cuenta cuantas entradas coinciden con los parámetros dados.

Parámetros

in	*e*	entrada.

Devuelve

Como el conjunto no puede tener entradas repetidas, devuelve 1 (si se encuentra la entrada) o 0 (si no se encuentra).

Postcondición

no modifica el conjunto.

5.1.4.7. template<typename T, typename CMP> bool conjunto< T, CMP >::empty ( ) const

Chequea si el conjunto esta vacio (size()==0)

Postcondición

No se modifica el conjunto.

5.1.4.8. template<typename T, typename CMP> conjunto::iterator conjunto< T, CMP >::end ( )

end del conjunto

Devuelve

Devuelve un iterador al final del conjunto (posicion siguiente al ultimo).

Postcondición

no modifica el conjunto.

5.1.4.9. template<typename T, typename CMP> iterator conjunto< T, CMP >::erase (const iterator *position*)

Borra una entrada en el conjunto . Busca la entrada y si la encuentra la borra.

Parámetros

in	*val*	entrada a borrar.
in	*position*	itarador que apunta a la entrada que geremos borrar

Devuelve

devuelve la posicion siguiente al elemento borrado (para la version con iterador) o el numero de elementos borrados

Postcondición

Si esta en el conjunto su tamaño se decrementa en 1.

5.1.4.10. template<typename T, typename CMP> size_type conjunto< T, CMP >::erase (const value_type & *val*)

5.1.4.11. template<typename T, typename CMP> iterator conjunto< T, CMP >::find (const value_type & *s*)

busca una entrada en el conjunto

Parámetros

in	*s*	entrada a buscar.

Devuelve

Si existe una entrada en el conjunto con ese valor devuelve el iterador a su posicion, en caso contrario devuelve iterador al final de conjunto

Postcondición

no modifica el conjunto.

5.1.4.12. template<typename T, typename CMP> const_iterator conjunto< T, CMP >::find (const value_type & *s*) const

5.1.4.13. template<typename T, typename CMP> pair<iterator,bool> conjunto< T, CMP >::insert (const value_type & *val*)

Inserta una entrada en el conjunto.

Parámetros

val	entrada a insertar

Devuelve

un par donde el segundo campo vale true si la entrada se ha podido insertar con éxito, esto es, no existe una mutación con igual valor en el conjunto. False en caso contrario. El primer campo del par devuelve un iterador al elemento insertado, o end() si no fue posible la insercion

Postcondición

Si e no esta en el conjunto, el size() sera incrementado en 1.

5.1.4.14. template<typename T, typename CMP> pair<iterator,bool> conjunto< T, CMP >::insert (value_type & *val*)

5.1.4.15. template<typename T, typename CMP> iterator conjunto< T, CMP >::lower_bound (const value_type & *val*)

busca primer elemento que no está por debajo ('antes', '<') de los parámetros dados.

Parámetros

in	*val*	entrada.

Devuelve

Devuelve un iterador al primer elemento que cumple que "elemento<e" es falso, esto es, el primer elemento que es mayor o igual que val Si no existe devuelve end

Postcondición

no modifica el conjunto.

5.1.4.16. template<typename T, typename CMP> const_iterator conjunto< T, CMP >::lower_bound (const value_type & *val*) const

5.1.4.17. template<typename T, typename CMP> conjunto& conjunto< T, CMP >::operator= (const conjunto< T, CMP > & *org*)

operador de asignación

Parámetros

in	*org*	conjunto a copiar.

Devuelve

Crea y devuelve un conjunto duplicado exacto de org.

5.1.4.18. template<typename T, typename CMP> size_type conjunto< T, CMP >::size ( ) const

numero de entradas en el conjunto

Postcondición

No se modifica el conjunto.

Devuelve

numero de entradas en el conjunto

5.1.4.19. template<typename T, typename CMP> iterator conjunto< T, CMP >::upper_bound (const value_type & *val*)

busca primer elemento por encima ('después', '>') de los parámetros dados.

Parámetros

in	*val*	entrada. Devuelve un iterador al primer elemento que cumple que "elemento>e", esto es, el primer elemento ESTRICTAMENTE mayor que val Si no existe devuelve end

Postcondición

no modifica el conjunto.

5.1.4.20. template<typename T, typename CMP> const_iterator conjunto< T, CMP >::upper_bound (const value_type & *val*) const

5.1.5. Documentación de los datos miembro

5.1.5.1. template<typename T, typename CMP> CMP conjunto< T, CMP >::comp [private]

5.1.5.2. template<typename T, typename CMP> vector<value_type> conjunto< T, CMP >::vm [private]

La documentación para esta clase fue generada a partir del siguiente fichero:

- conjunto.h

6. Documentación de archivos

6.1. Referencia del Archivo conjunto.h

```
#include <string>
#include <vector>
#include <iostream>
#include "mutacion.h"
```

Clases

- class conjunto< T, CMP >
  Clase conjunto.

Funciones

- template<typename T , typename CMP >
  ostream & operator<< (ostream &sal, const conjunto< T, CMP > &C)
  imprime todas las entradas del conjunto

6.1.1. Documentación de las funciones

6.1.1.1. template<typename T , typename CMP > ostream& operator<< (ostream & *sal,* const conjunto< T, CMP > & *C*)

imprime todas las entradas del conjunto

Postcondición

No se modifica el conjunto. Implementar tambien esta funcion

6.2. Referencia del Archivo documentacion.dox

Índice alfabético