Prácticas Conjuntos Disjuntos

Especificación del TAD Conjunto Disjuntos

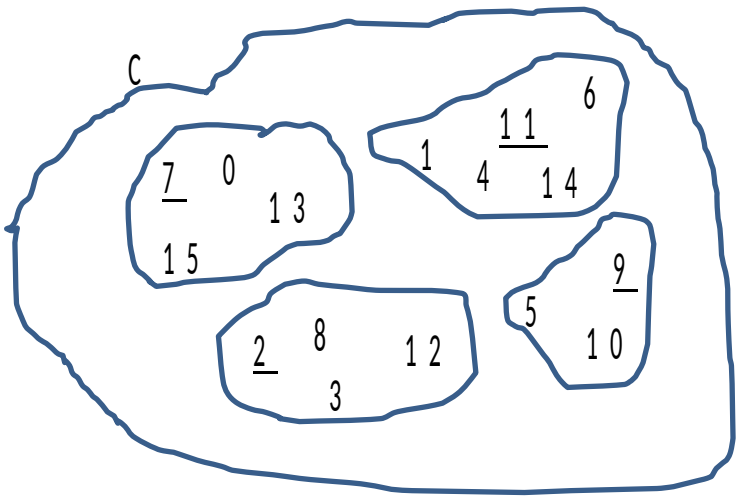
Estructura de datos que mantiene una colección $\boldsymbol{P}=\{C_1,C_2,...,C_n\}$ de conjuntos disjuntos entre sí, cuyos elementos son los enteros consecutivos $U=\{1, 2, \ldots, N\}$, y cada conjunto C_i representa una clase de equivalencia según alguna relación *R*. Al conjunto de **subconjuntos disjuntos** que representan las clases de equivalencia, y cuya unión es U, se le denomina **Partición** de U.

La operaciones básicas para este tipo abstracto de datos son:

> **crear(P)**: inicia la estructura partición P a una situación inicial donde todas las relaciones son falsas excepto las reflexivas.
>
> **buscar(x,P)**: devuelve la clase de equivalencia a la que pertenece el elemento x en la partición P.
>
> **unir(C_x,C_y,P):** establece una relación de equivalencia entre el elemento x e y, es decir, une los conjuntos que representan las clases de equivalencia de x y de y en la partición P.

Teniendo en cuenta estas especificaciones, deben implementarse las operaciones básicas según diferentes representaciones, como se indica en los enunciados que se detallan en la página siguiente.

Para la implementación consideraremos el conjunto $U=\{0, 2, \ldots, 15\}$ y formaremos las clases de equivalencia que se muestran en la siguiente figura



-figura 1-

1. Representación mediante MATRICES

Se representa la partición mediante una matriz de tamaño N almacenando en cada celda el representante de su clase de equivalencia

P[i]=j ⇒ el elemento i ∈ a la clase de equivalencia cuyo representante es j

AYUDA: Se proporcionan como ayuda los ficheros con los tipos y prototipos (**conjuntos.h**), la implementación de algunas funciones auxiliares para ver el contenido de la partición y de las clases de equivalencia (**conjuntos.c**) y un fichero de prueba que genera a partir de la situación inicial (todas las relaciones son falsas excepto las reflexivas) las clases de equivalencia de la figura 1 (**prueba.c**).

2. Representación mediante LISTAS ENLAZADAS

Cada clase de equivalencia se representa mediante una lista enlazada que contiene sus elementos. Representamos la partición mediante una matriz de tamaño N cuyos elementos son de tipo lista, utilizando para la representación de listas una estructura con dos punteros, uno al primer elemento y otro al último de la lista que representa la clase de equivalencia

P[i].primero contiene la dirección de la primera celda de la lista cuyo representante es i

P[i].último contiene la dirección de la última celda de la lista cuyo representante es i

AYUDA: Se proporcionan como ayuda los ficheros con los tipos y prototipos (conjuntos.h), la implementación de algunas funciones auxiliares para ver el contenido de la partición y de las clases de equivalencia (conjuntos.c) y un fichero de prueba que genera a partir de la situación inicial (todas las relaciones son falsas excepto las reflexivas) las clases de equivalencia de la figura 1 (prueba.c).

3. Representación mediante ÁRBOLES CON PUNTEROS AL NODO PADRE

Cada clase de equivalencia se representa por un árbol, utilizando su raíz para nombrar al conjunto. Para representar los árboles solo necesitamos un puntero al padre de cada nodo, por ello seguimos utilizando una matriz donde:

Si P[i]=i entonces i es a la vez el nombre del conjunto y su raíz

Si P[i]=j≠ i entonces j es el padre de i en algún árbol

Notas

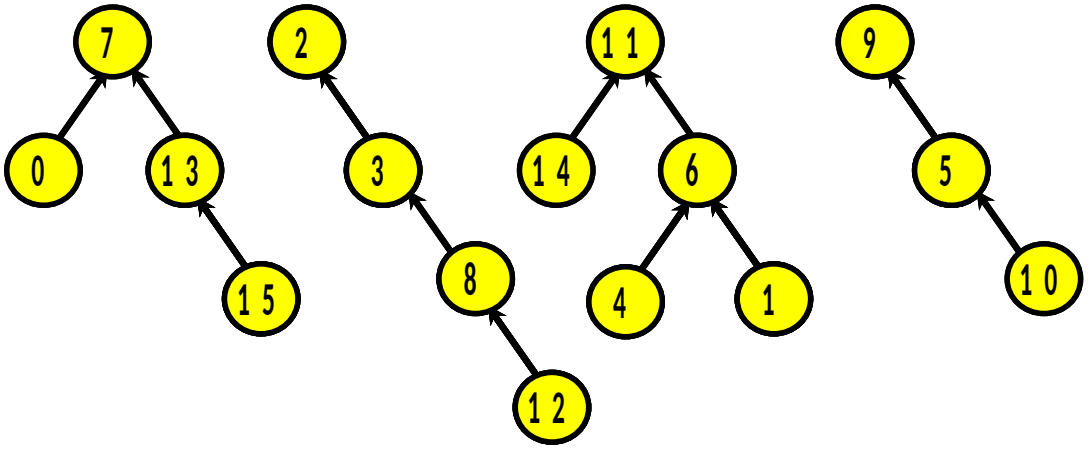
(1) La inclusión del 0 como elemento del conjunto U obliga a modificar el criterio de representación respecto al utilizado en teoría.

(2) Utilizamos un criterio arbitrario para **unir(C_x,C_y,C):** la nueva raíz es x

(3) Los dos primeros parámetros de entrada del operador unión tienen que ser los

representantes del conjunto,es decir, las raíces de los árboles que representan su clase de equivalencia.

(4) Utilizando la secuencia de uniones del programa de prueba se obtiene los siguientes árboles (figura 2) para representar las clases de equivalencia



-figura 2-

AYUDA: Se proporcionan como ayuda los ficheros con los tipos y prototipos (**conjuntos.h**), la implementación de algunas funciones auxiliares para ver el contenido de la partición (**conjuntos.c**) y un fichero de prueba que genera a partir de la situación inicial (todas las relaciones son falsas excepto las reflexivas) las clases de equivalencia de la figura 1 (**prueba.c**).

4. Unión por TAMAÑO

Mejorar el comportamiento de la búsqueda modificando la operación unión de forma que el árbol más grande sea la raíz del nuevo árbol. Cambia el convenio de representación:

Si ***i*** es la raíz de un árbol **C[i]** contiene el negativo del tamaño de su árbol

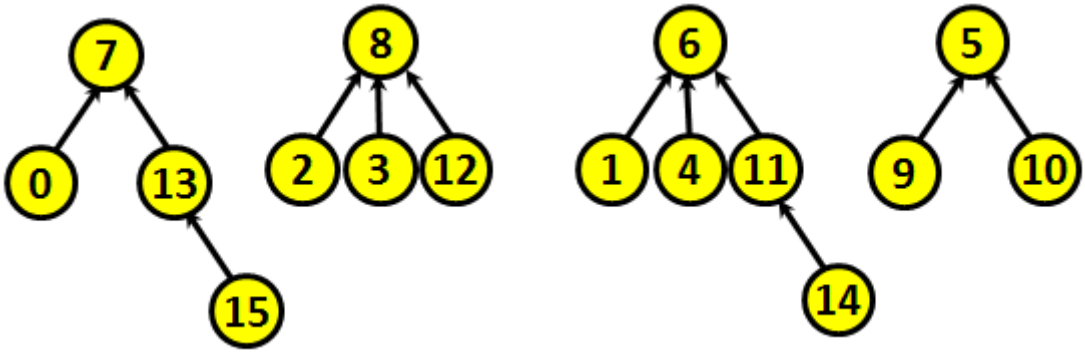
Si P**[i]=j>0** entonces ***j*** es el padre de ***i*** en algún árbol

Notas:

(1) Si los árboles tienen el mismo tamaño en **unir(C_x,C_y,P)** la nueva raíz es x.

(2) Los dos primeros parámetros de entrada del operador unión tienen que ser los representantes del conjunto,es decir, las raíces de los árboles que representan su clase de equivalencia.

(3) Utilizando la secuencia de uniones del programa de prueba se obtiene los siguientes árboles (figura 3) para representar las clases de equivalencia.



-figura 3-

5. Unión por ALTURA

Mejorar el comportamiento de la búsqueda modificando la operación unión de forma que el árbol de más alto sea la raíz del nuevo árbol. Cambia el convenio de representación:

Si **i** es una raíz P**[i]** contiene el negativo de la altura de su árbol

Si P**[i]=j>0** entonces ***j*** es el padre de ***i*** en algún árbol

Notas

(1) La inclusión del 0 como elemento del conjunto U obliga a modificar el criterio de representación respecto al utilizado en teoría. Consideraremos que en la situación inicial los árboles tienen altura -1 en lugar de cero y, que en general si i es una raiz C[i] contiene, en negativo, la altura mas uno de su árbol.

(2) Si los árboles tienen la misma altura en **unir(C_x,C_y,P)** la nueva raíz es x

(3) Los dos primeros parámetros de entrada del operador unión tienen que ser los representantes del conjunto,es decir, las raíces de los árboles que representan su clase de equivalencia.

(4) ¿Qué modificaciones son necesarias en el programa de prueba para obtener los mismos árboles, para representar las clases de equivalencia, que en unión por tamaño (figura 3)?

6. Compresión de CAMINOS

Mejorar el comportamiento de la operación búsqueda implementando la compresión de caminos: una vez que se conoce la raíz, se regresa por el camino de búsqueda, modificando el padre de cada nodo del camino, para que sea directamente la raíz.

Ayuda para la realización de la práctica:

Se proporcionan los ficheros correspondientes a los tipos y prototipos para las diferentes representaciones (conjuntos.h) y algunas funciones auxiliares que permiten mostrar las clases de equivalencia y el contenido de la partición (conjuntos.c). Estos ficheros se adjuntan en carpetas diferentes según la representación elegida: matrices (carpeta conjuntosM), listas (carpeta conjuntosL) y árboles (carpeta conjuntosA).

Se proporciona un único fichero de prueba (**prueba.c**) que puede utilizarse con las tres implementaciones. La compilación condicional de este fichero de prueba, tal como se muestra en el fichero **makefile**, permite al usuario utilizar el TAD Conjunto utilizando las diferentes representaciones. En este fichero de prueba, cuando la representación elegida es mediante árboles, en la operación unión debe tenerse en cuenta que los argumentos de entrada coincidan con la raíz del árbol (representante de la clase de equivalencia). Esto puede asegurarse sustituyendo **unir(C_x,C_y,P)** por ***unir(buscar(x,P), buscar(y,P),P)***. ¿Ocurre también cuando la representación es mediante listas?

En general, para **asegurar** que las relaciones de equivalencia de los elementos se establecen de forma adecuada, la operación unión, en **todas** las representaciones, debe utilizarse con los representantes de las clases de equivalencia de los elementos que relaciona:

unir(buscar(x,P), buscar(y,P),P)

Normas generales

En la realización de las prácticas se deben seguir los siguientes criterios:

1. Es **obligatorio** utilizar los tipos y prototipos siempre que se proporcionan en el fichero cabecera. Si no se proporcionan debe crearse el fichero cabecera correspondiente o añadir al proporcionado los prototipos de todas las funciones que se implementen.
2. La codificación de las funciones se realizará en un fichero fuente diferente al del programa principal.
3. Para visualizar el correcto funcionamiento de las operaciones implementadas se debe crear un fichero de prueba desde el cual se llamará a las funciones implementadas.
4. Se debe crear el correspondiente Makefile para la correcta creación del fichero ejecutable.