Técnicas Avanzadas de Programación:
Práctica 2ª:
El Reto:
Conjuntos y distancia mínima

Acedo Legarre, Aitor
Faci Miguel, Santiago

Agosto de 2004

Powered by LaTeX.

Contents

1 Introducción al problema

El objetivo de la práctica era el de proponer una implementación para el tipo de datos *conjunto de números naturales distintos* con las operaciones de *insertar*, *borrar*, *buscar*, que además incluirá la operación de *distancia mínima* definida en el enunciado de la práctica.

2 Solución propuesta

Para conseguir la implementación del conjunto de números naturales distintos nos decidimos por la utilización de dos tipos de datos familiares para nosotros como son los árboles AVL y los montículos, la idea es utilizar un árbol AVL para almacenar los números contenidos en el conjunto, gracias a esta estructura conseguimos que las operaciones de inserción , búsqueda y borrado se encuentren implementadas de una manera bastante optimizada. Para una implementación eficiente de la operación de la distancia mínima recordamos el uso de los montículos para encontrar en un tiempo constante el número menor o mayor dentro de un conjuto de números, lo cual es de una inestimable utilidad para nuestra implementación.

En el montículo almacenamos las distancias de un elemento del conjunto a su siguiente, ya que para encontrar la distancia mínima del conjunto simplemente tenemos que almacenar las distancias entre los elementos más próximos entre sí y la menor de ellas será la distancia mínima del conjunto. Esta idea requiere que modifiquemos ligeramente el contenido almacenado en el árbol AVL, en las implementaciones corrientes de este tipo abstracto de dato simplemente se almacena en los nodos el fac-

tor de balance del nodo, el valor contenido en el nodo, la clave y los punteros a los subárboles izquierdo y derecho, ahora deberemos añadir también dos punteros más, uno que apunta al nodo que almacena el dato anterior en el conjunto y otro que apunta al dato siguiente en el conjunto, lo cual constituye una lista doblemente enlazada de todos los elementos del conjunto.

El montículo también ha sufrido algunas ligeras variaciones sobre los algoritmos descritos en algorítmica en la página web de la asignatura de Estructuras de datos y algoritmos. Una de las más claras variaciones es el desarrollo del procedimiento *elimina_item*, el cual elimina del montículo el elemento que le pasamos como argumento, este no es el método de borrado que se suele utilizar en los montículos comunes y ha sido implementado por nosotros.

La otra modificación que hemos introducido ha sido el añadir un puntero en el elemento del montículo que apunta al nodo del árbol al cual pertenece la distancia en cuestión, este puntero es necesario ya que cuando borramos alguna de las distancias del montículo, dicho borrado puede hacer varias las posiciones dentro del montículo de alguna de las distancias, por lo que el cambio de posición también tiene que quedar reflejado en el campo *i_dist* del nodo del árbol.

Una de las limitaciones que nos hemos encontrado al implementar el conjunto ha sido la longitud del montículo, hemos optado por utilizar la cantidad más grande de las series de pruebas que se iban a realizar a la práctica de las comentadas en el servidor web.

Tenemos que reseñar que la implementación de los árboles AVL es una modificación de la que el profesor Javier Campos ha colgado en la página web de la asignatura de Técnicas Avan-

zadas de Programación, hay que indicar también que ciertos algoritmos como el de inserción y el de borrado han sido modificados para que se adaptaran a las necesidades de nuestra implementación, los cambios realizados en el código vienen explicados con claridad en los lugares apropiados.

Los cambios mencionados tienen que ver con mantener la coherencia entre las dos estructuras utilizadas para implementar el conjunto, ya que es obvio que cuando, por ejemplo, borramos un elemento del conjunto tenemos que borrar el nodo correspondiente del árbol y ello hace que ciertas distancias se tengan que borrar y otras nuevas tengan que ser insertadas, gracias a que los algoritmos utilizados para gestionar el árbol y el montículo son de coste logarítmico como ya sabemos, esto hace que los costes para gestionar el conjunto también lo sean.

2.1 Coste Asintótico de las Operaciones

El coste asintótico de las operaciones está relacionado con los elementos que contiene el montículo, si suponemos que el montículo tiene n elementos entonces tendremos los siguientes costes:

- crear() tiene coste O(1).
- vaciar() tiene coste O(n).
- insertar() tiene coste O(log(n)).
- borrar() tiene coste O(log(n)).
- buscar() tiene coste O(log(n)).
- dm() tiene coste O(1).

2.2 Estructuras de datos utilizadas

2.2.1 Montículo

Definición para el heap:

```
subtype elemento is integer range 0 .. MAX_NUM;
--Tipo de los indices del montículo
    subtype indice is integer range 1 .. MAX_NUM;

--Identifica al tipo del nodo en el árbol AVL
    type nodo;
    type avl is access nodo;

    type nodo_heap is record

--La distancia entre el nodo y su sucesor
        d: natural;
--Puntero a un nodo del árbol al que
--pertenece la distancia
        e: avl;
    end record;

    type vector is array (indice) of nodo_heap;
    type min_heap is record
        v: vector;
        num: elemento;
    end record;
```

2.2.2 Árbol AVL

Definición para el arbol AVL:

```
    type factores_balance is (pesado_izq,
                          balanceado,
                          pesado_der);
    type nodo is record
--Elemento del conjunto
        val: natural;
    --Factor de balance del nodo
    balance: factores_balance;
    --hijo izquiero, hijo derecho
    izq, der: avl;
    --predecesor y puntero al sucesor
    pred, succ: avl;
    --Indice de la distancia de este nodo
--con su sucesor en el monticulo
    i_dist: elemento;
    end record;
```

2.2.3 Conjunto

Definición del tipo conjunto:

```
    type conj is record
--Raiz del árbol AVL
        raiz: avl;
--Montículo de distancias
        h: min_heap;
    end record;
```

3 Programas de prueba

El primer programa, *test_conj_dm*, que hemos implementado para realizar las pruebas se ha diseñado para ser lo más flexible posible, con este objetivo en mente no hemos creído oportuno tener un fichero de texto en el que se almacenarán los valores aleatorios que se iban a insertar en el conjunto, sino que lo que hacemos es pasar dos parámetros al programa, el primero de los cuales nos indica cuantos números generamos aleatoriamente para ser introducidos en el conjunto, y después le pasaremos el número de veces que ejecutará una de las operaciones al azar sobre el conjunto.

Existe una puntualización a este programa de prueba, el hecho de que sea tan flexible hace que la persona que lo utilice deba conocer unas ciertas cuestiones de la implementación. El problema en nuestro caso es que podamos introducir cualquier número de datos para almacenarlos en el árbol, esto influye en la implementación estática del montículo que hemos realizado, por lo que es importante tener en cuenta que el número que introduzcamos como valor para el parámetro *num_dat* deberá ser menor o igual que la longitud del montículo definida en el fichero *conj_dm.ads* en la constante *MAX_NUM*. Para solucionar el problema del tamaño del montículo se ha optado por asignar un tamaño del montículo de 100000 elementos que será el máximo de datos introducidos en las pruebas que van a ser realizados por el profesor.

Otro detalle a tener en cuenta será el rango que tengan los números aleatorios generados para ser introducidos en el árbol, ya que si el límite para la generación de estos números es sustancialmente más pequeño que el número de inserciones que el usuario le ha indicado al programa que haga, por el hecho de

que no introducimos números repetidos en el árbol, el número de inserciones efectivas será igual a la cantidad de números diferentes generados por el programa de prueba. Para evitar que haya problemas en las pruebas y en los programas que pudieran realizarse para el código del conjunto con distancia mínima se ha dejado visible la constante *MAX_NUM* así cualquier programa de prueba podrá conocer cual es el tamaño actual del montículo. Nuestro programa de prueba tiene como límite máximo para generar números aleatorios el mismo número que da la capacidad al montículo.

El segundo programa que denominamos *test_estatico_conj_dm*, lo que hace es generar una muestra de 10 números aleatorios entre 1 y 10 para que sean insertados en el arbol y posteriormente lo que hacemos es borrar en orden inverso como se han generado para comprobar que tanto el borrado como la inserción funcionan correctamente.

El número de elementos generados es tan pequeño para que se pueda apreciar en pantalla el resultado de la ejecución de la prueba, además conforme se van borrando los números pintamos en pantalla la lista de sucesores de los elementos que quedan en el conjunto.

4 Modo de funcionamiento

Con ánimo de facilitar la tarea de la generación de los ejecutables para probar el funcionamiento de la práctica hemos creído oportuno añadir un fichero *makefile*, que automatiza totalmente este proceso, simplemente tendremos que escribir *make*, sin ninguna opción, en la ubicación donde hayamos descomprimido los fuentes y automáticamente la aplicación se generará, solamente nos quedará

ejecutar los ejecutables generados.

Este proceso cuenta además con una serie de opciones que vamos a comentar a continuación:

- *make test_estatico*: compilará el fichero para el test que hemos definido como estático.
- *make test*: compilará el fichero del test no estático.
- *make clean*: borrará los ficheros *.o *.ali .
- *make clean_estatico*: borra el ejecutable del test estatico.
- *make clean_test*: borra el ejecutable del test.
- *make clean_all*: borra todo a la vez.
- *make exportar*: modifica la variable de entorno TMPDIR por si hiciera falta en la compilación.

5 Bibliografía

- Hilliam, B.
  Introduction to abstract data types using Ada
  Prentice Hall International Editions

- Barnes, J.
  Programming in Ada 95
  Addison Wesley

- GNU libavl (a collection of binary search and balanced tree library routines)