Práctica 1 MARP
Montículo de Fibonacci

Autor: Javier Cortés Tejada

1. Guía sobre la práctica

La implementación de la estructura de datos se encuentra en el fichero FibonacciHeap.java. Ahí se detalla el funcionamiento de todas las operaciones con suficientes comentarios.
De cara a comprobar el correcto funcionamiento de dicho estructura se ha generado una pequeña GUI la cual puede ser lanzada desde el fichero Main.java para realizar las pruebas pertinentes. Puntualizo que en el enunciado de la práctica no se exigía dicha interfaz para la entrega, sino que se ha desarrollado para la verificación del funcionamiento de la misma (pues hacer ejecuciones paso a paso a mano resulta bastante inoperativo y poco visual), por lo tanto, las explicaciones de los ficheros que la implementan no están elaboradas con tanto rigor como los de la propia estructura de datos, aunque funciona perfectamente.

2. Notas

- La ejecución GUI implementada usa un modelo vista controlador (MVC) independiente de la implementación de la estructura de datos, luego no es necesario ver el código de la interfaz para entender la implementación de dicha estructura.
- El fichero Plots.java se ha usado para generar las gráficas de tiempos de las distintas operaciones disponible. Para volver a generar dichas gráficas basta con lanzar el método main de ese fichero (cuidado con el valor de la constante N que determina el número de ejecuciones de cada método, pues en el caso de las operaciones con coste en O(1) se han llegado a hacer $5x10^6$ operaciones, pero en las de coste O(log n) conviene no pasar de $1x10^6$ operaciones ya que tardan bastante en ejecutarse).
- En la GUI, el primer campo para introducir datos indica el la clave del nodo a insertar, decrecer, ect, mientras que el segundo solo se usa en decrecer clave indicando el nuevo valor del nodo.
- Se ha incluído el código de todos los ejemplos de este documento en el fichero Pruebas.java, de tal manera que si se quisiesen ejecutar basta con descomentar el código indicado dentro del método main, pues usar los códigos de otros ejemplos no va a dar resultados coherentes con lo expuesto aquí.

3. Ejemplos de ejecución

3.1. Introducción

Para todos los ejemplos de ejecución se va a usar como base el siguiente fragmento de código en Java, el cual se incluirá en el proyecto por si fuese necesario comprobar si los resultados mostrados en este apartado son coherentes con dicho código (también se incluirán las ejecuciones realizadas en cada ejemplo en el fichero Pruebas.java):

```
1  public static void main(String[] args) {
2      HashMap<Double, FibonacciNode> map = new HashMap<Double,
          FibonacciNode>();
3      FibonacciHeap heap = new FibonacciHeap();
4      FibonacciNode node = null;
5
6      for (double i = 5; i <= 14; i++) {
7              node = new FibonacciNode((int) i);
8              map.put(i, node);
9              heap.insert(node);
10     }
11
12     heap.removeMin();
13     map.remove(5);
14     heap.decreaseKey(node, 1.0);
15     map.remove(14);
16     map.put(1.0, new FibonacciNode(1));
17
18     System.out.println(heap.nodeVisualization());
19     System.out.println(heap.treeVisualization());
20
21 }
```

Este código da como resultado los siguientes logs por consola:

```
1
6
7
|—8
|—9
|   |—10
|—11
|   |—12
|   |—13
```

donde tenemos que 1, 6 y 7 son nodos raíz dentro de nuestro montículo, mientras que 8 y 9 son hijos de 7, 10 lo es de 9 y así sucesivamente para el resto de nodos en función de su representación.

Node = [parent = —, key = 1, degree = 0, right = 7, left = 6, child = —, mark = false]
Node = [parent = —, key = 7, degree = 3, right = 6, left = 1, child = 8, mark = false]
Node = [parent = —, key = 6, degree = 0, right = 1, left = 7, child = —, mark = false]
Node = [parent = 7, key = 8, degree = 0, right = 11, left = 9, child = —, mark = false]
Node = [parent = 7, key = 11, degree = 2, right = 9, left = 8, child = 12, mark = false]
Node = [parent = 7, key = 9, degree = 1, right = 8, left = 11, child = 10, mark = false]
Node = [parent = 9, key = 10, degree = 0, right = 10, left = 10, child = —, mark = false]
Node = [parent = 11, key = 12, degree = 0, right = 13, left = 13, child = —, mark = false]
Node = [parent = 11, key = 13, degree = 0, right = 12, left = 12, child = —, mark = true]

En esta otra parte del log tenemos la información detallada de cada nodo, la cual no se ha representado en el esquema de arriba.

3.2. Insertar

La operación de inserción en esta estructura tiene coste en O(1), pues simplemente se trata de una reasignación de punteros para introducir el nuevo nodo en la lista de raíces. También se reasigna el puntero al mínimo, pues este puede cambiar de valor tras una inserción, aunque esta operación también esta en O(1).

3.2.1. Caso A: insertar un nodo con clave menor que el mínimo

Insertamos en nodo con valor 0 y el montículo queda con la siguiente forma:

0
1
6
7
|—8
|—9
| |—10
|—11
| |—12
| |—13

En cuanto a la información de los nodos solo se ha añadido una línea nueva en referencia al nodo con clave 0, y se ha modificado la información de los nodos 1 y 7, pues se han reasignado sus punteros tras la inserción:
Node = [parent = —, key = 0, degree = 0, right = 7, left = 1, child = —, mark = false]
Node = [parent = —, key = 7, degree = 3, right = 6, left = 0, child = 8, mark = false]
Node = [parent = —, key = 1, degree = 0, right = 0, left = 6, child = —, mark = false]

3.2.2. Caso A: insertar un nodo con clave menor que el mínimo

Insertamos en nodo con valor 100 y el montículo queda con la siguiente forma:

1
6
7
|—8
|—9
| |—10
|—11
| |—12
| |—13
100

En este caso han tenido lugar las mismas modificaciones que en el ejemplo anterior, pero no se ha reasignado el mínimo pues no ha sido necesario:
Node = [parent = —, key = 1, degree = 0, right = 100, left = 6, child = —, mark = false]
Node = [parent = —, key = 100, degree = 0, right = 7, left = 1, child = —, mark = false]
Node = [parent = —, key = 7, degree = 3, right = 6, left = 100, child = 8, mark = false]

3.3. Eliminar mínimo

Esta operación se divide en dos partes:

- En la primera se toman todos los hijos del mínimo, introduciéndolos en la lista de raíces y éste se elimina.
- En la segunda se *consolida* el montículo para restaurar el invariante, de manera que se procesa la lista de raíces *linkando* aquellos nodos con el mismo número de hijos para obtener un montículo sin raíces con el mismo grado.

Cabe destacar que el coste amortizado de esta operación está en O(log n), pues la consolidación implica recorrer la lista de raíces al completo, donde el tamaño de esta lista es logarítmico en función del mayor rango de todos los nodos del montículo.

Para el ejemplo vamos a insertar un nodo con clave 100 y vamos a ejecutar 3 veces la operación borrar mínimo pues dado el ejemplo de partida, las dos primeras aplicaciones de esta operación *no hacen nada*, pues solo borran dos nodos de la raíz:
8
|—100
|—9
| |—10
|—11
| |—12
| |—13

Viendo el resultado obtenido, ahora tenemos un montículo con 1 raíz de grado 3. Si hacemos la operación paso a paso, antes de la tercera llamada a *consolidar* nos queda un montículo con la siguiente forma:
8
9
|—10
11
|—12
|—13
100

Aquí vemos que los hijos del mínimo (7) han pasado a formar parte de la raíz y tenemos 3 raíces con grados distintos salvo los nodos 8 y 100, ambos con grado 0. La tercera llamada a *consolidar* provoca los siguientes cambios:

- Aplica la operación *link* a 8 y 100 por tener el mismo grado, devolviendo una raíz de grado 1.

8
|—100
9
|—10
11
|—12
|—13

- Aplica de nuevo la misma operación a los nodos 8 y 9 al tener el mismo grado, devolviendo un nodo raíz de grado 2.

8
|—100
|—9
| |—10
11
|—12
|—13

- Por último aplica de nuevo *link* sobre 8 y 11 al tener el mismo grado, devolviendo un nodo con grado 3.

3.4. Decrecer clave

Esta operación, al igual que la de insertar, también esta en O(1) pues se trata de reasignaciones de punteros.

3.4.1. Caso A: decrecer clave y cortar nodo

Aplicamos decrecer clave sobre el nodo 10, partiendo del caso base expuesto anteriormente. Esto nos devuelve el siguiente resultado:
0
1
6
7
|—8
|—9
|—11
| |—12
| |—13
Como podemos comprobar, en nodo con clave 10 ha pasado a tener clave 0 y en nodo con clave 9 ha perdido un hijo, luego ha quedado marcado como se puede comprobar en su información interna:
Node = [parent = 7, key = 9, degree = 0, right = 8, left = 11, child = —, mark = true]

3.4.2. Caso B: decrecer clave de un nodo hijo con padre marcado

Aplicamos un decrecer clave al nodo 11 para partir del caso propicio para el ejemplo, donde tanto el nodo 13 como su padre (11) están marcados:
0
1
6
7
|—8
|—9
| |—10
|—11
| |—13

Node = [parent = 11, key = 13, degree = 0, right = 13, left = 13, child = —, mark = true]
Node = [parent = 7, key = 11, degree = 1, right = 9, left = 8, child = 13, mark = true]

Aplicamos de nuevo decrecer clave sobre 13 y obtenemos el siguiente log: 0
1
6
7
|—8
|—9
| |—10
2
11

En este caso podemos comprobar los efectos de la operación *corte en cascada*, pues al decrecer la clave del nodo 13 a 2, éste ha sido cortado y añadido a la raíz, y al estar marcado el padre (11), éste también se ha añadido a la raíz por el *corte en cascada*.

4. Bibliografía

Referencias

[1] Thomas H Coemen, Charles E. Leiserson, Ronald L. Rivest y Clifford Stein, *Introduction to Algorithms*, tercera edicion, Cambridge, Massachusetts London, England.