Pablo Martín Huertas

Doble Grado Ingeniería Informática-Matemáticas

Implementación detallada de montículos de Fibonacci.

Universidad Complutense de madrid

Visión general

La principal motivación de la creación de la estructura conocida como un montículo se basa en permitir al usuario llevar un conjunto de parejas que contienen prioridad y un registro. Lo más importante de dicha estructura es permitir el acceso a la mínima prioridad y sus datos en todo momento. Para ello se pretende buscar que las operaciones sobre el montículo tenga unos costes del orden mínimo posible.

En esta búsqueda de costes mínimo han surgido varias implementaciones distintas de los montículos, entonces la pregunta ahora es ¿Qué tienen de particular los montículos de Fibonacci?

La mejor forma de entender la eficiencia de los montículos de Fibonacci es comparando sus costes con los de otras implementaciones distintas. Por ejemplo, tomemos unos de los montículos más usados en la actualidad como pueden ser los binomiales y analicemos las diferencias en sus costes:

operación	mont. binomial (caso peor)	mont. Fibonacci (**amortizado**)
crear vacío	$\Theta(1)$	$\Theta(1)$
insertar	$\Theta(\log n)$	$\Theta(1)$
mínimo	$\Theta(\log n)$	$\Theta(1)$
borrar mínimo	$\Theta(\log n)$	$\Theta(\log n)$
unión	$\Theta(\log n)$	$\Theta(1)$
reducir clave	$\Theta(\log n)$	$\Theta(1)$
borrar	$\Theta(\log n)$	$\Theta(\log n)$

Esta tabla proporciona una evidente mejora en los costes. Los montículos de Fibonacci se van a caracterizar en que son capaces de mantener una gran parte de las operaciones en modo constante; ya que, su filosofía se basa en usar la operación “borrar mínimo” para ser capaz de consolidar el montículo, es decir recolocar las parejas del conjunto de forma que las demás operaciones puedan tener un acceso constante.

Una vez vista la motivación de la implementación de los montículos de Fibonacci, vamos a presentar su estructura interna para después analizar las operaciones que nos permite hacer en detalle entendiendo así de donde vienen los costes de la tabla.

Estructura Interna

La implementación se divide en dos capas. La que el usuario usa directamente: ‘’ColeccionFib.h’’ y una implementación interna de los montículos individuales llamada ‘’ImpColeccFib.h’’. La separación se debe a la necesidad de tener un mapa global a una colección de montículos para poder así mantener operaciones como la de unir con un coste constante.

Los montículos van a componerse de una estructura elemental llamada: Nodo.

Se trata de un struct que contiene:

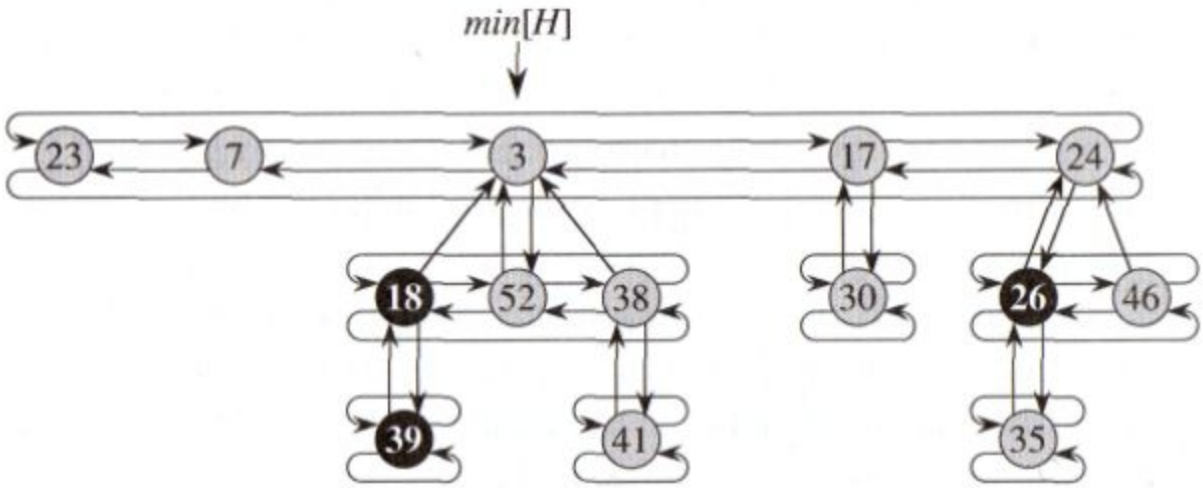
-4 punteros: dos al elemento siguiente y anterior y otros dos al padre y al hijo.

-Una pareja que contiene un double con la prioridad del nodo y un registro que puede instanciar el usuario para añadir los datos que desee.

Además existirá un puntero principal denominado mínimo que en todo momento apuntará al nodo de menor prioridad.

Finalmente, hay un “unordered_map” en la colección (tabla hash implementada en visual studio) que será capaz de obtener un puntero a un Nodo en un tiempo constante conociendo únicamente su prioridad.

Veamos un ejemplo de un montículo:



Tal y como están dispuestos se observa que a cada nivel de profundidad los nodos se juntan formando una estructura de lista enlazada.

Veamos ahora las operaciones que proporciona el montículo de Fibonacci.

Operaciones

Las operaciones se ofrecen desde la fachada “ColeccionFib.h”; la cual, está disponible a el usuario.

Los costes vienen dados respecto al número de elementos del montículo.

1. **ColeccionFib()** (Coste constante) Crea una colección de montículos vacía.
2. **crearMonticulo(string id)** (Coste constante) Crea un montículo vacío en la colección que realiza la llamada.
3. **bool isEmpty(string id)** (Coste constante) Busca el montículo con id adecuado y devuelve true si está vacío y false en caso contrario. Para ello basta comprobar si mínimo está apuntando a un nodo.
4. **void insert(string id, pair<double, T> pareja)** (Coste constante) Inserta un nuevo nodo en el montículo "id" con la prioridad del double un registro instanciable T. El nodo directamente se coloca a la izquierda del nodo mínimo.
5. **void deleteMin(string id)** (Coste logarítmico) Busca el montículo id y borra el mínimo de manera constante y llama a la operación auxiliar consolidar. Ésta es la que va a tener coste logarítmico y su objetivo es recorrer la lista doblemente enlazada principal y va calculando la altura de los distintos nodos. A su vez siempre que encuentre dos nodos con hijos con la misma altura los juntará en un solo árbol con una altura de una unidad adicional.
6. **void decreaseKey(string id, double prioridad, double nuevaPrioridad)** (Coste constante) Busca la clave con "prioridad" en el unordered_map de "id". Una vez localizado el nodo, lo corta situándolo a la izquierda del mínimo y marca al padre. Si el padre ya estaba marcado se repite el corte de manera recursiva.
7. **void deleteKey(string id, double prioridad)** (Coste logarítmico) Busca el nodo con "prioridad" en el mapa del montículo "id". Le atribuye una nueva prioridad que es la del mínimo -1 (usando decreaseKey) y acto seguido llama a deleteMin, el cual le proporciona el coste logarítmico.
8. **pair<double,T> findMin(string id)** (Coste constante) Devuelve la pareja del nodo marcado por el mínimo del montículo "id".

9. **void fibUnion(string idA, string idB)** (Coste constante) Une el montículo con identificador idA al montículo idB. Desde la unión en adelante ambos identificadores se refieren al mismo montículo.

10. **~ColeccionFib()** (Coste lineal) Destructora que se deshace de la estructura interna de la colección.

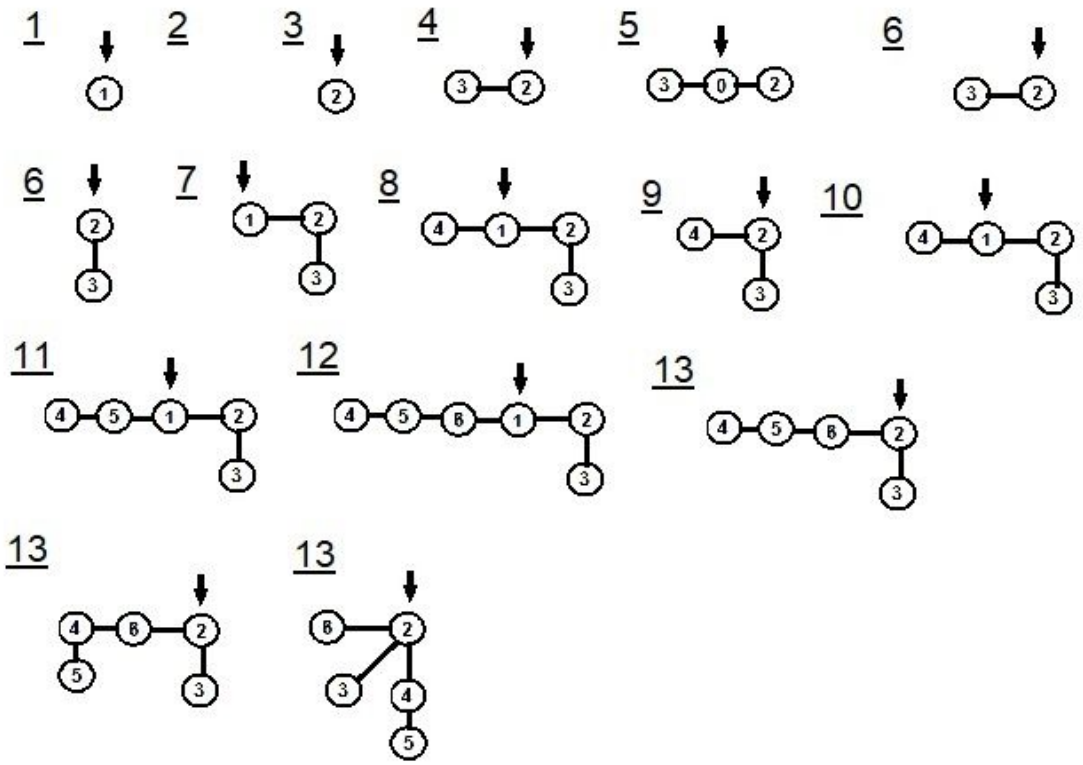
Corrección de las operaciones

Para asegurarnos de que los algoritmos funcionan correctamente he realizado una serie de tests (se encuentran en el archivo Source.cpp proporcionado) para comprobar casos con todas las operaciones ofrecidas.

Las estructuras resultantes de cada uno de los tests han sido comprobadas exhaustivamente en modo de depuración de visual studio. Se recomienda tener el código del cpp al lado mientras se siguen los cambios en las figuras (creadas personalmente con un programa de edición) para verificar que efectivamente todo funciona correctamente.

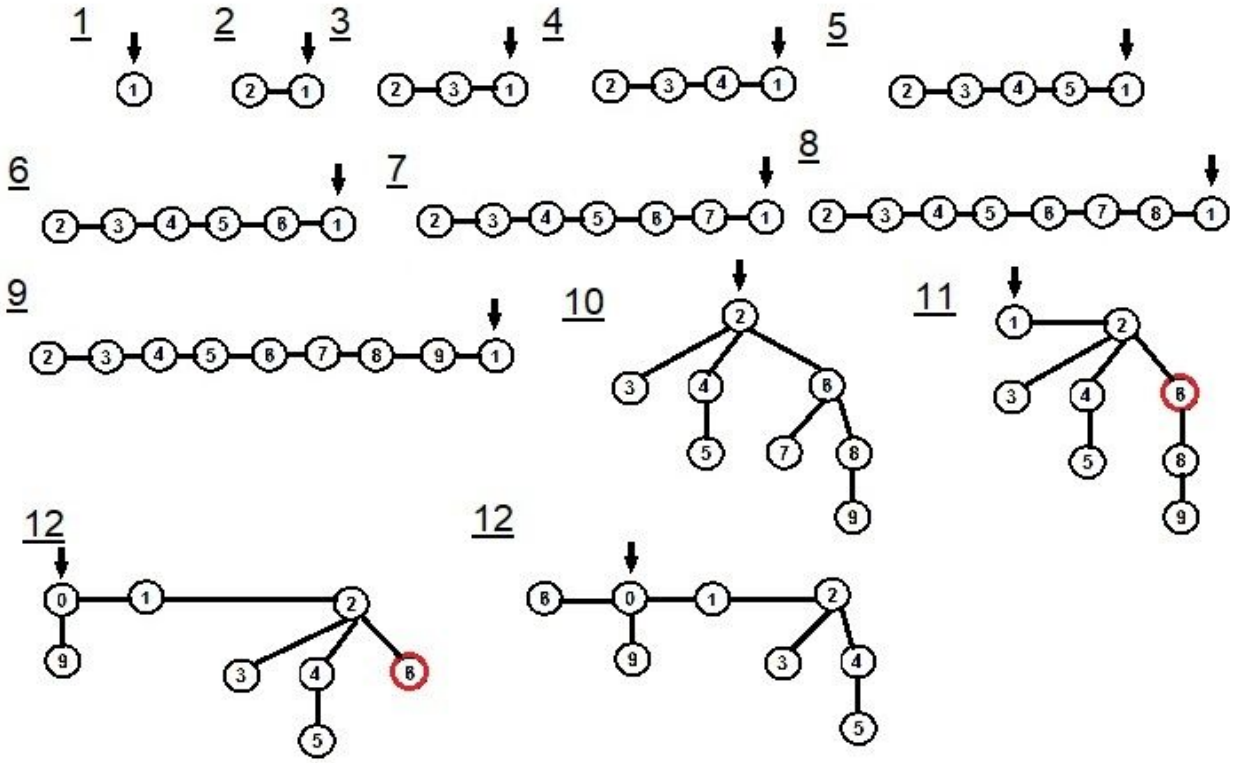
1. Test (inserciones y borrados del mínimo consolidando)

   En este test se realizarán 13 operaciones siendo la última la más extensa (para ello en la figura siguiente se puede ver que la operación 13º ha sido fraccionada en tres pasos para entender el resultado)



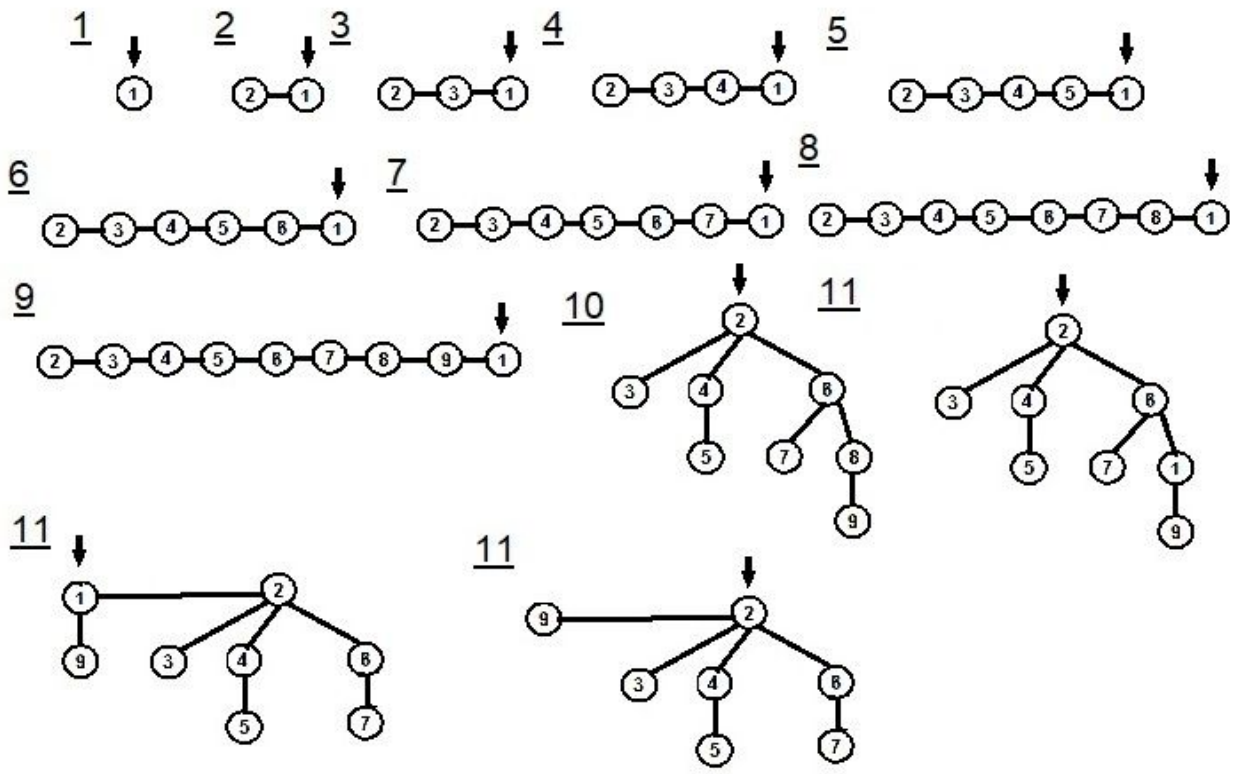
2. Test (Decremento de claves y corte en cascada)

En este caso nos fijamos principalmente en la operación 12 la cual debe realizar la llamada recursiva que realiza el corte adicional.



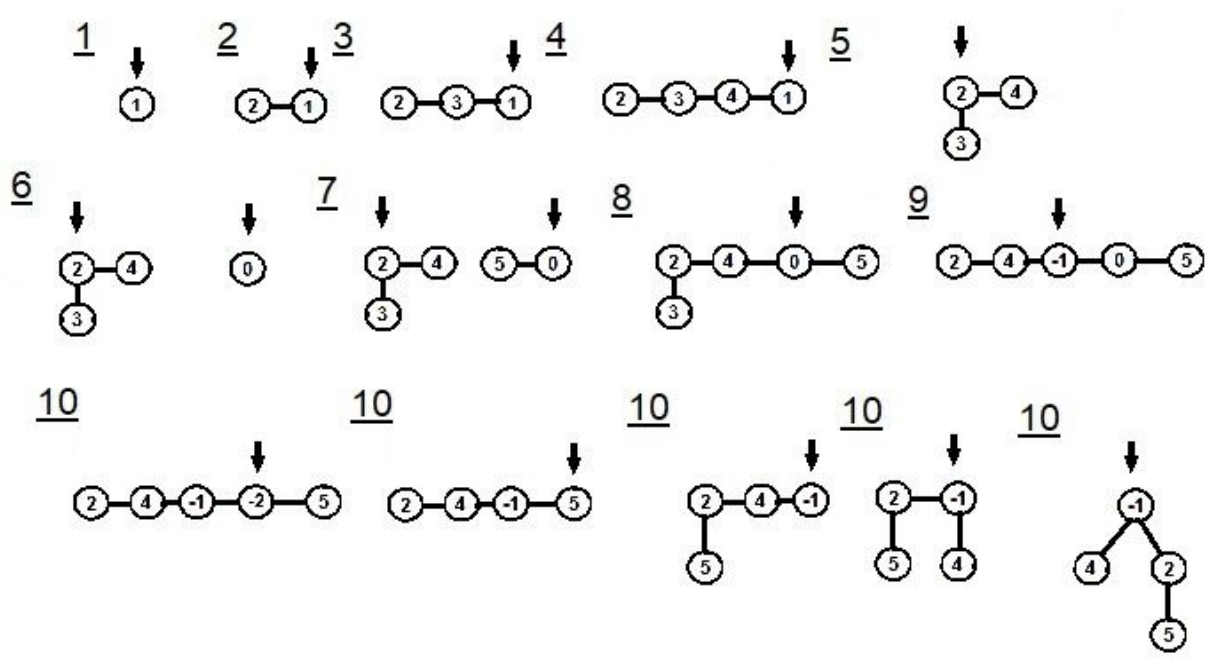
3. Test (Eliminación de una clave concreta)

La operación deleteKey se realiza en la décimo primera y última instrucción. Su desarrollo se compone de el cambio a la prioridad del mínimo -1 (en este caso 2-1 = 1) y de la llamada a deleteMin.



4. Test (Unión y operaciones posteriores)

Para acabar realizamos una unión de los montículos y comprobamos que al aplicarle operaciones adicionales sigue respondiendo de manera correcta.



Esto concluiría el repaso de los montículos de Fibonacci, desde el motivo de su creación hasta las pruebas de casos concretos con el código proporcionado. Como apunte adicional cabe destacar que el código de los fichero ColeccionFib.h e ImpColeccFib.h están repleto de comentarios los cuales a pesar de que añaden extensión a la implementación (Al final han sido ~500 líneas) ayudan a comprender qué se está haciendo en cada momento.

Webgrafía:

https://www.cs.princeton.edu/~wayne/teaching/fibonacci-heap.pdf

http://www.cs.princeton.edu/~wayne/cs423/lectures/fibonacci-4up.pdf

https://cv4.ucm.es/moodle/pluginfile.php/4641971/mod_resource/content/1/monticulosUnion.pdf (apuntes del campus UCM)

http://webdiis.unizar.es/asignaturas/TAP/material/3.4.fibonacci.pdf

https://en.wikipedia.org/wiki/Fibonacci_heap

https://www.geeksforgeeks.org/fibonacci-heap-set-1-introduction/

https://brilliant.org/wiki/fibonacci-heap/