MÉTODOS ALGORÍTMICOS EN RESOLUCIÓN DE PROBLEMAS

MONTÍCULOS DE FIBONACCI

2 de abril de 2019

Autor: Jorge Villarrubia Elvira
Universidad Complutense de Madrid
Facultad de Informática

Contenidos

1. INTRODUCCIÓN

1.1. Aclaraciones previas

Comenzamos esta sección dando una visión global de la implementación escogida, así como del por qué de la misma. En los sucesivos apartados se explicará cada aspecto con mucho más detalle.

La implementación que se adjunta, se corresponde con una familia de montículos de Fibonacci desarrollada en C++. Toda ella está alojada en el mismo archivo de encabezado (.h), con el objetivo de facilitar al usuario la tarea de incorporación de la misma. Basta con que incluya este fichero a su proyecto para hacer uso de la estructura.

Por contra, se tiene un código algo menos claro, sobretodo en lo que a vinculación entre clases se refiere, que si se hubiera modularizado. Por ello es muy importante seguir con atención la siguientes subsecciones.

1.2. Clases y relación entre las mismas

La estructura consta fundamentalmente de dos clases: ***familiaFib*** para la familia de montículos y ***mFib*** para cada montículo concreto.

La idea es que la clase familia sea una especie de 'gestor' de montículos, que por tanto queda jerárquicamente por encima de cada uno de ellos.

Además, para la representación de los elementos del montículo y la interconexión entre los mismos en forma de listas doblemente enlazadas, se ha incluido una estructura *'struct'* dentro de la familia. Dicha estructura constituye un ***Nodo***.

```
struct Nodo;
        using Link = Nodo * ;
        struct Nodo {
                T elem;
                Link hIz, hDer, hijo, padre;
                bool marca; // Indicara la perdida de un hijo de este nodo
                size_t grado; // Altura del subarbol que tiene por debajo
        };
```

Cada objeto ***Nodo*** contiene a su vez los punteros necesarios para que el montículo organice un conjunto de ellos como una lista doble. Esto es, enlaces a su *'hermano izquierdo'* ***(hIz)*** y *'hermano derecho'* ***(hDer)***. Y los necesarios para conectar las listas a distintas alturas del montículo, es decir, al *'padre'* ***(padre)*** y al hijo ***(hijo)***.

Asimismo, tiene atributos ***marca*** de tipo **bool** y ***grado*** de tipo entero no negativo, que indican la pérdida de un hijo y la altura del subárbol que hay por debajo, respectivamente. Y por supuesto, el elemento de tipo **T** genérico ***(elem)*** que representamos con el nodo.

Nótese que en el fichero este *'struct'* lleva 2 constructores que se han obviado en el código anterior. Uno con recibe todos los atributos y otro que solo recibe el elemento y pone punteros a *'null'*, la marca a *'false'* y el grado a 0.

Puesto que el usuario no interactúa con los montículos, sino con la familia, todos los métodos y atributos de la clase *mFib* son **protegidos** o **privados**. Además, aunque a priori pueda chocar, se ha optado por desarrollar *mFib* como clase interna de *familiaFib*, con el fin de ocultar esta primera clase al usuario.

Por último, para que dicha familia pueda acceder a los antes mencionados métodos y atributos de los montículos, se ha declarado en ellos a la familia como ***clase amiga*** o ***friend class***.

2. DETALLES DE LA IMPLEMENTACIÓN

2.1. Constructores

Dado que añadir diferentes constructores que se adapten a las necesidades del usuario no entraña ninguna dificultad, apenas se ha implementado uno 'vacío' con el que bastará para probar la estructura. Esto se ha hecho tanto para la familia pública, como para el montículo interno. De forma que se deja la adición de nuevos constructores al usuario interesado.

Constructor de *mFib*

```
protected:
        mFib(Comparator c = Comparator()) : antes(c), min(nullptr), nelems(0) {}
```

Constructor de *familiaFib*

```
public:
        familiaFib(Comparator c = Comparator()) : antesFam(c), nelemsTotal(0) {}
```

Ambos reciben un comparador (parámetro de sus clases), que por defecto es el ***less<T>*** del tipo instanciado. Y dado que en la familia llevaremos la cuenta del total de elementos en todos los montículos de la misma y en cada montículo la cuenta de los almacenados en él en particular, estos constructores ponen sus números de elementos y punteros a valores iniciales 0 y *'null'*
respectivamente.

2.2. Inserción

Para la inserción se han considerado dos casos. Que el usuario quiera insertar un elemento en la familia sin indicar donde o bien que mediante un entero (≥ 1) indique el montículo concreto donde desea hacerlo.

Inserción sin concretar

```
void insertar(T const& e){
        if (!elementos.count(e)) {
                mDir m = new mFib<T>(antesFam);
                Link x = m->insertar(e);
                monticulos.insert({ monticulos.size() + 1, m });
                elementos.insert({ e, {x, m} });
                nelemsTotal++;
        }
}
```

Inserción concreta

```
void insertar(T const& e, int pos) {
        if (pos > monticulos.size() || pos <= 0) throw domain_error("El monticulo
            donde querias insertar no existe. Consulta los disponibles (desde 1 hasta
            size())");
        else if (!elementos.count(e)) {
                Link x = monticulos[pos]->insertar(e);
                nelemsTotal++;
                elementos.insert({ e, {x, monticulos[pos]} });
        }
}
```

Resulta relevante apreciar que no se admiten duplicados. Y por eso antes de cada inserción primero se comprueba que no esté ya el elemento en cuestión. Para ello se utiliza un diccionario no ordenado ***elementos***, que se actualizará pertinentemente en la familia.

La inserción no concreta se basa en la creación de un montículo unitario nuevo en la familia, de forma que se llama al *insertar* de *mFib* con este nuevo montículo. Mientras que en la concreta se hace la llamada con el montículo solicitado (si existe), tomándolo de otro diccionario no ordenado ***montículos*** que almacena punteros (tipo *mDir*) a montículos.

Función insertar del montículo

```
Link insertar(T const & e) {
        Link x = new Nodo(e);
        insertaEnLPrincipal(x);
        nelems++;
        return x;
}
```

Esta última función *protected* es la que, recibiendo aún el tipo **T**, crea el objeto **Nodo** en memoria. Además hace una llamada a *insertaEnLPrincipal*, una función auxiliar de relevancia, que realmente hace el trabajo 'sucio' de toqueteo de punteros para añadir el nodo y que se aconseja leer en el .h con detenimiento. Por razones lógicas aquí no se entrará más en ella, aunque sí que se volverá a mencionar en operaciones posteriores.

2.3. Devolución del mínimo

Esta es sin duda la operación más simple de la estructura. Teniendo en cuenta que cada montículo tiene como atributo un puntero a su elemento mínimo, lo devolvemos sin más.

Únicamente hay que considerar que como la familia no almacena un mínimo de todos sus montículos, el usuario debe indicar de cuál de ellos desea recibir el mínimo. Por lo que se debe comprobar la existencia del mismo y en caso de conflicto lanzar la correspondiente excepción.

Función mínimo de la familia

```
T const& minimo(int pos) const {
        if (pos > monticulos.size() || pos <= 0) throw domain_error("El monticulo
            donde querias insertar no existe. Consulta los disponibles (desde 1 hasta
            size())");
        else { try {return monticulos[pos]->minimo();}
                        catch (domain_error d) {throw;}
        }
}
```

Función mínimo del montículo

```
T const& minimo() const {
        if (min == nullptr) throw domain_error("El monticulo esta vacio");
        else return min->elem;
}
```

Apréciese que la función de la familia relanza la excepción de dominio controlada por el montículo, en caso de que se pida el mínimo sin haber elementos.

2.4. Unión de montículos

La unión de montículos tampoco presenta una gran dificultad. Se reduce a la concatenación de las listas principales de los implicados.
Para ello, el usuario nos indicará dentro del rango de montículos válidos los que desea unir y se hará una llamada sobre uno de ellos pasando el otro como parámetro. De esta forma la unión se hace sobre el primero y en el segundo se colocará un nuevo montículo vacío.

Operación de unir en la familia

```
void unir(int pos1, int pos2) {
        if (pos1 > monticulos.size() || pos1 <= 0 ) throw domain_error("El monticulo
            1 al que querias unir, no existe...");
        if (pos2 > monticulos.size() || pos2 <= 0) throw domain_error("El monticulo 2
            que querias unir, no existe...");
        if(pos1 == pos2) throw domain_error("No puedes unir un monticulo consigo
            mismo. No admitimos claves duplicadas");
        monticulos[pos1]->unir(*(monticulos[pos2]));
        monticulos[pos2] = new mFib<T>(antesFam);
}
```

Como de costumbre, controlamos que los montículos elegidos para la operación sean correctos informando oportunamente del error y nos aseguramos de no unir montículos que puedan dar lugar a claves repetidas, siendo la unión de un montículo consigo mismo el único caso posible.

2.5. Extraer el mínimo

Para mantener un buen coste en esta operación (log(n)), teniendo en cuenta la simplicidad y eficiencia de otras operaciones como la inserción, es necesario complicarse considerablemente. Esta complejidad reside en la función privada ***consolidar*** de cada montículo.

La función ***quitarMinimo*** de la familia, con que interactúa el usuario, apenas comprueba los habituales errores de dominio y llama a la del montículo donde se encuentre.

Por ello pasamos directamente a analizar la función del montículo, donde se empieza por guardarse el mínimo y su hijo y para hacer un recorrido que permita añadir todos los hijos del mínimo a la lista principal.

Añadir los hijos del mínimo a la lista principal

```
Link minimo = min;
Link hijo = min->hijo;
if (hijo != nullptr) {
        minimo->hIz->hDer = hijo;
        hijo->hIz->hDer = minimo;
        Link anteriorHijo = hijo->hIz;
        hijo->hIz = minimo->hIz;
        minimo->hIz = anteriorHijo;
        min = minimo;
        minimo->hijo = nullptr;
        for (Link act = hijo; act != minimo; act = act->hDer) act->padre = nullptr;
}
```

Esto se ha hecho para ahora eliminar el mínimo simplemente moviendo punteros de la lista doble principal para que 'desaparezca' y que no por ello desaparezcan sus hijos, sólo el mínimo.

Desaparición del mínimo de la lista principal

```
minimo->hIz->hDer = minimo->hDer;
minimo->hDer->hIz = minimo->hIz;
delete minimo;
```

Obsérvese que aunque en el código anterior se haya incluido el *delete* del *Nodo* en memoria dinámica, en el código original se hace bastante después por la necesidad de acceder a la variable *minimo* tras su 'desaparición'.

Ahora, si el hermano derecho del mínimo es él mismo, la lista principal resultante será vacía y por tanto el nuevo mínimo será inexistente *(nullptr)*. En caso contrario se coloca el puntero a mínimo *min* momentáneamente al hermano derecho del anterior mínimo y se cede el resto de trabajo a *consolidar*.

2.5.1. Operación de consolidar

Esta es sin duda la operación más compleja de toda la estructura, por lo que se recomienda atender con atención la siguiente explicación. Se trata de una operación privada propia de cada montículo concreto. Como ya se viene haciendo, se adjuntarán trozos clave del código que serán comentados después.

Vector de raíces a montículos según su grado

```
size_t D = 2 * ceil(log(double(nelems)));
vector<Link> A;
```

Para empezar, resulta absurdo consolidar un montículo vacío, por lo que, para ahorrarse problemas, el código de la función solo se ejecuta en caso contrario.

Una vez hemos considerado esto, se crea el vector que almacenará punteros a los distintos elementos de lista principal, almacenándose cada uno en la posición que indica su grado (número de hijos). En este proceso, tal y como queda demostrado en múltiples libros que estudian esta estructura, las raíces pueden alcanzar a lo sumo grado $\log_2(n)$, de forma que se instancia el vector con dicho tamaño.

'Colgado' entre raíces del mismo grado ordenadas en el vector

```
do { size_t g = act->grado;

    while (A[g] != nullptr) {
        y = A[g];
        if (act->elem > y->elem){
                guardaAct = act;
                act = y;
                y = guardaAct;}
        if (actSig == min && y == min) {
                actSig = y->hDer;}
        if (y == min) min = y->hDer;
             mezclar(y, act);
            A[g] = nullptr;
                    g++;
        }

    A[g] = act;
    act = actSig;
    actSig = act->hDer;} while (act != min);
```

Seguidamente, mediante la anidación de bucles *while*, se recorre la lista de raíces del montículo con el bucle de fuera y para cada una de ellas, haciendo uso del vector, se cuelga de otra con el mismo grado, o se le cuelga a ella otra raíz con el mismo grado. De esta forma, al salir no hay dos raíces con los mismos grados. Es decir, dos raíces que hubieran de ocupar la misma posición en el vector.

Un aspecto primordial en este proceso es el intercambio de raíces con el vector cuando ambas son iguales y la del vector tenga 'prioridad' según el orden establecido sobre la otra, para lo cual nos hemos ayudado de un puntero auxiliar ***guardaAct***.

También es importante tratar la condición de parada del recorrido, para la cual se ha de ser muy cuidadoso con el avance de los punteros. Para esto, se lleva otro puntero auxiliar ***actSig***, que nos indica la siguiente raíz a visitar. La parada se produce cuando estamos 'volviendo' a visitar el mínimo, por lo que vamos actualizando este a la raíz siguiente de la raíz 'colgada' para mantener dicho mínimo en la lista principal en una 'posición' aún por visitar evitando así un posible bucle infinito.

Un último caso raro en el avance es que el nuevo hijo sea el que hasta ahora era mínimo y además sea el siguiente en visitarse para lo cual se visitará después 'el siguiente del siguiente'. De lo contrario, el puntero que nos marca la siguiente visita indicaría una raíz que no está en la lista principal para la siguiente iteración. Nótese que esto se corresponde con la condición ***if (actSig == min and y == min)***, que ha de comprobarse antes de actualizar el mínimo.

Finalmente, en cada iteración del bucle interno se llama, ya con los punteros a las raíces adecuadas, a una función privada ***mezclar*** que 'cuelga' una raíz respecto a la otra y ya fuera de los bucles, una vez acabado todo esto, se hace un último recorrido de la lista principal para recalcular el mínimo.

2.6. Decrecimiento de una clave

Para decrecer una clave cualquiera del montículo manteniendo unos costes adecuados al resto de operaciones, es necesario tener acceso constante a todos lo elementos almacenados del mismo. Para ello, como ya se ha comentado anteriormente, se lleva un mapa no ordenado que se ha ido actualizado pertinentemente en las operaciones precedentes.

En la operación de la familia, nos limitamos a comprobar la existencia del valor a decrementar, haciendo uso del mencionado mapa, e impedimos que se decrezca la clave a una ya existente, puesto que no admitimos duplicados. Para todo ello lanzamos las correspondientes excepciones.

Función decrecer clave del montículo

```
if (antes(eAntiguo, eNuevo)) throw invalid_argument(error);
else {
        Link eCambiar = elementos[eAntiguo];
        eCambiar->elem = eNuevo;
        Link padre = eCambiar->padre;
        if (padre != nullptr && antes(eCambiar->elem, padre->elem)) {
                cortar(eCambiar, padre);
                cortarEnCascada(padre);
        }
        if (antes(eCambiar->elem, min->elem)) min = eCambiar;
```

Sin embargo, en la función monticular, tras comprobar que lo que pretende hacer el usuario es un decremento e impedir lo contrario, se hace un 'corte' con el nodo padre al decrementado y se insta a un 'corte' sucesivo en cascada, siempre que el cambiado se haya convertido en menor al padre. En las operaciones posteriores se explica con más detalle en qué consisten estos 'cortes'. Asimismo, se actualiza correspondientemente el diccionario de elementos.

2.6.1. Corte con el nodo padre

El antes mencionado 'corte' consiste en provocar la desaparición del nodo hijo recibido como argumento, que se corresponde con el decrecido en la operación pública que estamos abordando, para que deje de ser hijo de su padre. No tiene más misterio que un cuidado toqueteo de punteros que se adjunta a continuación.

```
hijo->hIz->hDer = hijo->hDer;
hijo->hDer->hIz = hijo->hIz;
padre->grado--;
insertaEnLPrincipal(hijo);
hijo->padre = nullptr;
hijo->marca = false;
```

2.6.2. Corte en cascada

Por otra parte el 'corte' en cascada no es más que la repetición del corte anterior hasta llegar de padre en padre a la raíz o a un padre ya marcado.
Así solo se realiza el 'corte' en los padres de marcados. Esto se ha hecho mediante una función recursiva, que además va marcando todos estos padres que van siendo menores que sus hijos. Recordemos, que cada *Nodo* tenía un atributo de tipo *bool* para llevar esta marca.

2.7. Borrado de un nodo cualquiera

En este último método ha vuelto a echar mano del acceso constante a cualquier elemento que nos proporciona el mapa no ordenado *elementos*, que además nos dice en qué montículo de todos los que tiene la familia, está el elemento a borrar. Tras comprobar que efectivamente se desea eliminar un elemento existente, se cede el trabajo a la función del montículo.

Función eliminar del montículo

```
void eliminar(T const& e) {
    if (nelems > 0) {
        if (e == min->elem) quitarMinimo();
        else{
                T minimo = min->elem;
                quitarMinimo();
                decrementarClave(e, minimo);
                quitarMinimo();
                insertar(minimo);
                }
        }
}
```

La idea del método del montículo es muy simple y se basa en reducir el problema a casos anteriores. Para lo que no hay inconveniente, ya que los costes hasta ahora han sido mejores o iguales que el propuesto para esta función, *O(log(n))*.

En caso de que haya elementos a eliminar, se comprueba si el que se ha pedido borrar es el mínimo, resolviéndose esto con una simple llamada al método ***quitarMinimo*** que ya hacía esta labor (apartado 2.5). En caso contrario, quitamos el mínimo (guardándolo), decrementamos la clave que queremos borrar al valor que tenía el mínimo mediante el método ***decrementarClave***, volvemos a quitar el mínimo y finalmente insertamos el mínimo original.

En este último caso nos hemos aprovechado de la no admisión de duplicados, ya que si quitamos el mínimo y decrecemos la clave a borrar al antiguo mínimo nos aseguramos de que esta clave es el nuevo mínimo (no podía haber dos mínimos), con lo que al volver a llamar a ***quitarMinimo*** quitamos la clave que queríamos, y luego simplemente tenemos que volver a meter el mínimo original que habíamos quitado, para dejar todo lo demás como estaba.

2.8. Otros métodos

Cabe mencionar la existencia de otros métodos en la familia, que están a entera disposición del usuario como ***creaMonticuloVacio*** o ***estaElemento*** que permiten, respectivamente, crear un montículo sin elementos en la familia para que la primera inserción no tenga que ser sin indicar el nº de montículo y consultar la existencia de un elemento en la familia.

Ambos tienen coste constante y han resultado de gran utilidad en las pruebas que se describirán con detalle en la próxima sección.

3. PRUEBAS

La metodología de pruebas escogida se apoya fundamentalmente en la aleatoriedad de la mismas y la comparación con estructuras que se sabe que son correctas. Para ello, siempre que ha sido posible, se han utilizado elementos obtenidos mediante el método ***rand***, de la biblioteca <*stdlib.h*> de C++, y se han comparado los resultados de nuestros montículos con los de la cola de prioridad de la STL.

3.1. Pruebas de los métodos

Para también probar la corrección del comparador y facilitar la lógica, con como está pensada la ***priority queue***, se ha utilizado en las pruebas el comparador *greater*<*T*>. Asimismo, para generar la semilla que utilizaremos en el método ***rand*** se ha usado el método ***time*** de la biblioteca <*time.h*>, pasándole un objeto nulo.

3.1.1. Prueba de inserción y obtención del mínimo

Para esta prueba simplemente se han insertado en ambas estructuras una gran cantidad de elementos aleatorios, pidiéndose el mínimo cada vez que se ha insertado uno.

Test de inserción y mínimo

```
srand(time(NULL));
priority_queue <int> pq;
familiaFib <int, greater<int>> fib;
fib.creaMonticuloVacio();
for (int i = 0; i < 200000; ++i) {
        int aleatorio = rand() %300000;
        pq.push(aleatorio);
        fib.insertar(aleatorio, 1);
        Assert::AreEqual(pq.top(), fib.minimo(1));
        }
}
```

Nótese que ese número suficientemente grande se ha fijado en 200.000 y que para que haya suficientes 'cambios' de mínimo como para que la prueba sea efectiva los números aleatorios generados se mueven en un rango suficientemente amplio: [0, 299.999].

Además, toda la inserción se hace sobre el mismo montículo (nº 1) de la familia, ya que la cola de prioridad con que comparamos no tiene pensada esta estructura de familia. Debemos pedir el mínimo siempre al mismo montículo.

3.1.2. Prueba de inserción y extracción del mínimo

Esta prueba es completamente análoga a la de la subsección anterior, a excepción de que aquí es necesario comprobar que el elemento que insertamos no está aún en el montículo. La cola de prioridad de la STL admite duplicados y nosotros no, y por tanto esta pudiera tener 2 mínimos, que en extracciones sucesivas darían una salida distinta a la nuestra. Obligamos pues a no tener repetidos en la cola de prioridad para hacer la prueba correctamente.

Test de inserción y extracción del mínimo

```
for (int i = 0; i < 200000; ++i) {
        int aleatorio = rand() %300000;
        if (!fib.estaElemento(aleatorio)) {
            pq.push(aleatorio);
                fib.insertar(aleatorio, 1);
                Assert::AreEqual(pq.top(), fib.minimo(1));
                pq.pop();
                fib.quitarMinimo(1);}
}
```

3.1.3. Prueba de decrecimiento de clave

Para probar el método que nos permite decrecer un elemento cualquiera del montículo se ha utilizado otra estructura de comparación, ya que las colas de prioridad de la STL carecen de una operación similar. Se ha hecho uso de una implementación de una cola de prioridad indexada, alojada en el fichero ***IndexPQ.h***, que también se sabe que es correcta.

Prueba de decrecimiento de clave

```
IndexPQ<int, greater<int>> pq(200000);
familiaFib <int, greater<int>> fib;
fib.creaMonticuloVacio();
for (int i = 0; i < 200000; ++i) {
int aleatorio = rand() %300000;
if (!fib.estaElemento(aleatorio)) {
        pq.push(aleatorio, aleatorio);
        fib.insertar(aleatorio, 1);
        }
}
for (int j = 0; j < 200000; ++j) {
        int aleatorioGrande = rand() %300000;
        int aleatorioPequeno = rand() %30000;
        if (fib.estaElemento(aleatorioGrande) && !fib.estaElemento(aleatorioPequeno)
            && aleatorioGrande < aleatorioPequeno) {
                pq.update(aleatorioGrande, aleatorioPequeno);
                fib.decrementarClave(aleatorioGrande, aleatorioPequeno);
                Assert::AreEqual(pq.top().prioridad, fib.minimo(1));
                }
        }
}
```

Básicamente se han insertado números grandes y aleatorios en ambas estructuras, representando en la cola indexada dicho número su prioridad, para después generar dos números (en la mayoría de casos el segundo más pequeño que el primero) que sean la clave a decrecer y el valor al que decrecer, respectivamente.

El primer número se genera entre 0 y 299.999, mientras que el segundo entre 0 y 29.999, con lo que es más probable que el primero sea mayor que el segundo, aunque no es seguro. Por tanto antes de actualizar los valores para después compararlos, comprobamos: que la clave a decrecer esté, que el nuevo valor que tomará no esté y que realmente sea un decrecimiento de clave.

3.1.4. Prueba de eliminación de clave cualquiera

Para esta prueba se ha tratado de comprobar la correcta eliminación de un número grande y aleatorio de elementos, usando de forma auxiliar un vector que nos permita almacenar en otra estructura dichos elementos insertados en el montículo, para luego eliminarlos, evitando excepciones y que hagan que el test falle.

Prueba de eliminación

```
srand(time(NULL));
vector<int> v;
familiaFib <int> fib;
fib.creaMonticuloVacio();
for (int i = 0; i < 200000; ++i) {
    int aleatorio = rand() %300000;
    if (!fib.estaElemento(aleatorio)) {
            v.push_back(aleatorio);
            fib.insertar(aleatorio, 1);
    }
}
while (!v.empty()) {
    fib.eliminar(v.back());
    v.pop_back();
}
Assert::AreEqual(true, fib.emptyRaiz(1));
}
```

Tras eliminar todos los elementos insertados, se llama a una función del montículo ***emptyRaiz*** que devuelve *true* en caso de que el mínimo haya quedado vacío. Comprobar con un 'empty' al uso que el número de elementos sea nulo podría tapar un error. El hecho de que el contador interno de elementos sea cero no es indicativo alguno de que se hayan eliminado realmente. Nuestro método comprueba que la raíz se nula, lo que resulta más fiable.

3.1.5. Prueba de unión

Para probar la unión interna de dos montículos de la familia, se han insertado números aleatorios en 3 montículos. Uno llevará los números pares generados, otro los impares y el tercero llevará tanto los pares como los impares, es decir, la unión de los 2 primeros. Se unen los primeros y se comprueba que sus elementos coincidan uno a uno con la que debería ser su unión, es decir el tercer montículo.

Prueba de unión de montículos

```
familiaFib <int, greater<int>> fibParesImpares;
familiaFib <int, greater<int>> fibTodos;
fibParesImpares.creaMonticuloVacio();
fibParesImpares.creaMonticuloVacio();
fibTodos.creaMonticuloVacio();
for (int i = 0; i < 200000; ++i) {
    int aleatorio = rand() %300000;
    if (aleatorio %2 == 0) fibParesImpares.insertar(aleatorio, 1);
    else fibParesImpares.insertar(aleatorio, 2);
    fibTodos.insertar(aleatorio, 1);
}
fibParesImpares.unir(1, 2);
while (fibTodos.size() > 0) {
    Assert::AreEqual(fibTodos.minimo(1), fibParesImpares.minimo(1));
    fibTodos.quitarMinimo(1);
    fibParesImpares.quitarMinimo(1);
    }
```

Nótese que para comprobarse la igualdad de montículos deseada se va extrayendo sucesivamente el mínimo de ambos, mientras este exista, verificándose su igualdad.

3.2. Comprobación del coste amortizado

Finalmente, para comprobar que el coste amortizado de la estructura, principal atractivo de la misma, es el deseado, se han realizado dos pruebas de tiempos intercalando operaciones. Ambas se encuentran en el fichero *CosteAmortizado.cpp* y pueden ser ejecutadas por consola.

En la primera de ellas, se han intercalado inserciones y extracciones del mínimo con una cierta frecuencia prefijada para lo segundo.

Prueba para el coste amortizado de inserción y extracción

```
    int t0 = clock();
    for (int i = 0; i < NUM_INS; ++i) {
            fib.insertar(i, 1);
            if (i %FREC_BORRADO == 0) fib.quitarMinimo(1);
    }
    int t1 = clock();
```

La función correspondiente a esta prueba se limita a insertar elementos desde el 0 hasta una constante prefijada suficientemente grande(*NUM INS*) como para que la prueba sea significativa y cada cierto número de ciclos también prefijado (*FREC BORRADO*) hace una llamada para extraer el mínimo. Se ha probado con un número de inserciones de 500.000 y una frecuencia de borrado de 3. Es decir, un total de 666.666 operaciones.

La función también se encarga de calcular en base a las constantes fijadas el número de operaciones, sacar el tiempo total en ser realizadas y dividir este segundo dato entre el primero (T(n)/n) para obtener el tiempo amortizado.

Con los datos antes mencionados se ha obtenido la siguiente salida por consola

```
Insercion y extraccion del minimo:
Numero de inserciones: 500000
Numero de extracciones: 166666
n: 666666
T(n) = 27.932
T(n) / n = 4.1898e-05
```

Si se cambian (aumenta significativamente) el número de operaciones, cambia drásticamente el tiempo que tardan en realizarse (se han llegado a probar con tamaños que tardan varios minutos), pero el coste amortizado (logarítmico) apenas varía subiendo de orden 10^{-5} a 10^{-4}.

De manera exactamente análoga se ha hecho con inserción y extracción de un elemento cualquiera. Esta combinación quizás goza de mayor interés pues el borrado de un elemento arbitrario hace a su vez una combinación de llamadas a extraer y decrementar clave, con lo se están también incluyendo estas en la prueba.

Salida de la segunda prueba

```
Insercion y eliminacion de elemento cualquiera:
Numero de inserciones: 500000
Numero de extracciones: 166666
n: 666666
T(n) = 38.107
T(n) / n = 5.71606e-05
```

Que como podemos apreciar es ligeramente superior a la anterior, ya que esta extracción tiene una constante más alta que la otra pues hace llamadas a varios métodos para su cometido, pero conservando el coste logarítmico. Su tiempo amortizado apenas varía en 10^{-5}.

4. BIBLIOGRAFÍA

Introduction to Algorithms.
Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest y Clifford Stein
Tercera edición. Capítulo 19
Disponible en: <https://labs.xjtudlc.com/labs/wldmt/reading%20list/books/Algorithms%20and%20optimization/Introduction%20to%20Algorithms.pdf>

Implementaciones de otras estructuras.
Alberto Verdejo
IndexPQ.h, PriorityQueue.h
Disponibles en el fichero del trabajo

Webs de consulta sobre C++.
Dominio público
cplusplus.com, stackoverflow.com, cppreference.com (fundamentalmente)