Universidad Complutense de Madrid

Montículos Binomiales

Métodos Algorítmicos En Resolución De Problemas

Facultad de Informática

Beatriz Herguedas Pinedo

Madrid Enero 2019

The first 90 percent of the code accounts for the
first 90 percent of the development time...
The remaining 10 percent of the code accounts
for the other 90 percent of the development time.

Abstract

En el presente trabajo se abordará la implementación detallada de los montículos binomiales, utilizando para ello el lenguaje de programación C++ y centrándonos sobre todo en la operación de unión de dos montículos dentro de la misma colección. Se añade también el estudio de los costes amortizados de las distintas operaciones implementadas, así como un estudio de la casuística y su prueba exahustiva.

Índice general

Capítulo 1

Introducción

Entre los distintos tipos de estructuras de datos existentes son destacables las colas de prioridad. La implementación de estas se puede realizar de distintas maneras. Dependiendo del enfoque que uno adopte, el coste amortizado obtenido puede variar notablemente, siendo el impacto de este tipo de datos abstracto de crucial importancia en múltiples situaciones. Una implementación pobre puede causar en una cola de prioridad que los costes de sus operaciones hagan inviable su uso en situaciones donde hay un gran número de datos por procesar. Las forma más común de abordar la implementación de una cola de prioridad se basa en el uso de montículos. Los montículos binomiales son un tipo de montículo utilizado para la implementación eficiente de colas de prioridad centrándose sobre todo en añadir la funcionalidad de unión de dos montículos, de la cual carecen los montículos binarios (el tipo más común de montículos utilizado a la hora de implementar colas de prioridad). En el año 1978, J. Vuillemin publicó un trabajo en el que se estudiaba detalladamente por primera vez este tipo de montículos [8]. En dicho trabajo, Vuillemin desarrolló un nuevo tipo de montículo que serviría para representar una colección de colas de prioridad, admitiendo las operaciones básicas de todo montículo (inserción, borrado, ...) y añadiendo la posibilidad de unir dos montículos dentro de la colección. Con todo, los montículos binomiales ofrecen una nueva forma de implementar, más eficientemente, una cola de prioridad, añadiendo además nuevas funcionalidades.

Capítulo 2

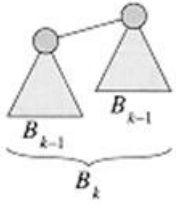
Definiciones

2.1. Árbol

Un árbol es un tipo abstracto de datos (TAD) que imita la estructura jerárquica de un árbol, con un valor en la raíz y subárboles representados como un conjunto de nodos enlazados, donde el número de hijos de cada nodo es variable, desde cero en el caso de una hoja, hasta un cierto número, al que llamamos grado del árbol. Además, cada nodo tiene un nivel asociado, siendo este cero para la raíz. El resto de nodos tienen un nivel más que el nivel de su padre. Así, la altura de un árbol es el máximo de los niveles de los nodos hoja (nodos sin hijos). [5]

2.2. Árbol binomial

El i-ésimo árbol binomial B_i, con $i \geq 0$, se define recursivamente como aquel que consta de un nodo raíz con i hijos, en donde el j-ésimo hijo, $1 \leq j \leq i$, es a su vez la raíz de un árbol binomial B_{j-1}. El árbol binomial B_i contiene 2^i nodos. [2]



2.3. Montículo

Un montículo no es más que una estructura de datos del tipo árbol con información perteneciente a un conjunto, que satisface la propiedad de montículo: si P es un nodo padre de C, entonces la clave de P es mayor o igual que la clave de C en un montículo de máximos, o menor o igual que la clave de C en un montículo de mínimos. [3] [1]

2.4. Montículo binomial

Un montículo binomial es una colección de árboles binomiales. Cada árbol binomial de la colección debe tener un tamaño diferente, y además todos ellos tienen que cumplir las propiedades del montículo. Para completar la definición, añadiremos punteros que unan cada raíz con la siguiente, por orden creciente de tamaño de los árboles binomiales. [2]

Capítulo 3

Aclaraciones Iniciales

Previamente a meternos de lleno en los montículos binomiales, es necesario introducir el tipo de implementación escogida y la estructura del código. [3]

En primer lugar, se ha implementado una familia de montículos en la clase binomialHeapFamily, desarrollada en C++, y escrita al completo en el mismo fichero, con el fin de facilitar la comodidad del usuario, aunque ello signifique tener un código más largo. Por ello se ha tenido un especial cuidado con el orden y la claridad en la escritura y comentarios.

Además, cada familia está formada por montículos de mínimos, donde el elemento en la raíz es menor que todos los elementos en el hijo izquierdo y en el derecho, y ambos hijos son a su vez montículos de mínimos. Así, la raíz del árbol coniene el mínimo de todos los elementos.Para ello hemos utilizado la clase interna binomialHeap, que corresponde a cada montículo en particular.

Uno de los principales atributos de un montículo es el nodo. Para ello hemos creado un struct con los elementos necesarios para implementar el montículo como lista enlazada. Esto es, punteros al padre, hijo y hermano de la derecha. Además, llevamos la clave del elemento y el grado del mismo. Por último, el nodo incluye dos constructores, uno vacío y otro con los atributos necesarios.

Por otro lado, en la clase binomialHeap tenemos tres atributos: un puntero al primer elemento del montículo, un puntero al elemento mínimo, y un entero que lleva el número de elementos del montículo.

```
struct Node {
        //Clave del elemento
        T key;
        //Puntero al padre, al hijo y al hermano
        Link parent, child, sibling;
        //Grado del elemento
        size_t degree
};
```

Por último, cada familia de montículos lleva un mapa con todos los elementos de la familia para tener acceso constante a ellos, y un mapa con los montículos (número y link al primer elemento), ya que cada uno va asociado a un número como forma de identificarlo, además del tamaño global de la familia.

Capítulo 4

Operaciones

En este capítulo veremos la implementación y explicación de cada una de las operaciones propias de un montículo binomial. [3] [4]

4.1. MakeHeap()

Se han implementado una serie de constructores tanto para la familia de montículos como para cada montículo en sí, que van a ser de mucha utilidad para la implementación del resto de operaciones. Esta operación tiene coste constante en todos los casos.

```
//Constructores de binomialHeap
private: binomialHeap(Link const& h) : head(h) {}

protected:
binomialHeap() : head(nullptr), nelems(0) {}

binomialHeap(T const& elem) : nelems(1) {
        head = new Node(elem);  min = head;
}

//Constructor de binomialHeapFamily
public: binomialHeapFamily() : totalElems(0){}
```

4.2. Insert()

En la operación de insertar hemos considerado dos casos: el caso en el que el usuario quisiera insertar un elemento sin indicar dónde, y el caso en el que indicase en qué montículo específico de la familia desea insertar el elemento. Veamos ambos casos por separado.

4.2.1. Insert(T e)

Este es el caso de inserción en el que el usuario quiere insertar un elemento sin indicar en qué montículo concreto, por lo que primero se comprueba que el elemento a insertar no esté ya (puesto que la familia de montículos está implementada de forma que no se admiten repetidos), y posteriormente se crea un nuevo montículo con el constructor y se inserta el elemento en él llamando a la función insert() sobre ese nuevo montículo. Finalmente se actualizan los diccionarios. Esta operación tiene coste logarítmico.

```
/*Inserta el elemento e en un montículo vacío*/
void insert(T const& e) {
  auto it = elements.find(e);
  if (it == elements.end()) {
        Heap H = new binomialHeap<T>;
        Link l = H->insert(e);
        heaps.insert({ heaps.size() + 1, H });
        elements.insert({ e,{l, heaps.size()} });
        totalElems++;
  }
}
```

4.2.2. Insert(T e, int n)

Este es el tipo de inserción en el que el usuario quiere insertar un elemento en un montículo concreto, por lo que se comprueba que dicho montículo sea correcto, que el elemento no esté ya, y se busca el montículo en cuestión. Posteriormente se inserta en dicho montículo llamando a la función

insert() sobre él. Finalmente se actualizan los diccionarios. Esta operación tiene coste logarítmico.

```
/*Inserta el elemento e en el montículo n*/
void insert(T const& e, int const& n) {
  if (n <= 0 || n > heaps.size())
        throw domain_error("This heap does not exist");
  auto it = elements.find(e);
  if (it == elements.end()) {
        Link l = heaps[n]->insert(e);
        elements.insert({ e, {l, n} });
        totalElems++;
  }
}
```

Ambas inserciones utilizan la función insert() de la clase montículo, que crea un montículo con un único elemento (el elemento que queremos insertar), y lo une al montículo donde queremos insertar utilizando la función privateUnion(), que une dos montículos de la misma familia, aunque esto lo veremos con detalle más adelante en la función unionWith().

```
Link insert(T const& e) {
    binomialHeap H = binomialHeap(e);
        privateUnion(H);
        nelems++;
        return H.head;
}
```

El motivo de que esta función sea privada es que es una función auxiliar que no debe estar disponible para ser utilizada fuera de la clase binomialHeap. Posteriormente a unir los montículos con el fin de insertar el elemento, se actualiza el número de elementos del montículo y se devuelve el link al montículo donde se ha insertado, lo cual podría ser últil para el usuario, y lo será también para nosotros en futuras operaciones.

4.3. Minimum(int n)

Esta operación devuelve el mínimo del montículo que el usuario solicite. Esta tarea es sencilla, pues como hemos dicho anteriormente, cada montículo lleva un atributo que es un link al elemento mínimo de dicho montículo. Esta operación tiene coste constante.

```
T const& minimum(int const& n) {
        if (n <= 0 || n > heaps.size())
                throw domain_error("This heap does not exist");
        else if (heaps[n]->empty())
                throw domain_error("Empty heap");
        return heaps[n]->minimum();
}
```

Primero se comprueba que el montículo del que se quiere saber el mínimo sea válido (es decir, pertenezca a la familia), y posteriormente se llama a la función minimum() de la clase montículo.

```
T minimum() const {
        return min->key;
}
```

4.4. ExtractMin(int n)

Esta función elimina el mínimo del montículo solicitado por el usuario, no sin antes comprobar que el montículo es válido. Para ello llamamos a la función extractMin() de la clase montículo, y devolvemos el mínimo que se ha eliminado, lo cual podría ser útil para el usuario. Esta operación tiene coste logarítmico.

```
T extractMin(int const& n) {
        if (n <= 0 || n > heaps.size())
                throw domain_error("This heap does not exist");
        if (heaps[n]->empty())
                throw domain_error("Empty heap");
        totalElems--;
        return heaps[n]->extractMin();
}
```

La función extractMin() de la clase montículo es una función larga donde se realiza toda la manipulación de punteros necesaria para eliminar el mínimo sin que se eliminen también sus hijos.

Esto se realiza eliminando primero el mínimo de la lista de raíces,

```
    T minim = min->key;
    Link y = min;
    Link prevx = nullptr;
    Link x = head;
    while (x != y) {
            prevx = x;
            x = x->sibling;
    }
    if (prevx == nullptr)
            head = x->sibling;
    else
            prevx->sibling = x->sibling;
```

invirtiendo la lista de hijos del mínimo,

```
Link z = nullptr;
Link w = y->child;
while (w != nullptr) {
        Link next = w->sibling;
        w->sibling = z;
        z = w;
        w = next;
}
```

y fusionando dicha lista con la de raíces.

```
binomialHeap H(z);
H.calculateMin();
calculateMin();
privateUnion(H);
calculateMin();
nelems--;
    if (nelems == 0) {
            min = nullptr;
            head = nullptr;
    }
return minim;
```

En esta tarea se utiliza la función unionPrivate() de la que hablaremos posteriormente, y la función calculateMin(), una función privada que calcula el mínimo de un montículo realizando la manipulación de punteros pertinente. Para más detalle aconsejo que se estudie detenidamente la implementación adjunta en el .h, pues por motivos evidentes no es posible añadirla aquí.

4.5. UnionWith(int h1, int h2)

Esta es sin duda la operación más complicada que tienen los montículos binomiales, así que prestémosle atención. Lo que hace la función es unir dos montículos que pertenecen a la familia de montículos. Esto lo hemos hecho así ya que, para poder unir elementos cualesquiera, tendríamos que actualizar el mapa donde llevamos todos los elementos de la familia, y eso se saldría del coste estipulado, por lo que una solución es permitir solamente unir montículos que ya pertenezcan a la familia. Para ello lo primero que se hace es, en efecto, comprobar que los dos montículos pertenecen a la familia y que no son el mismo, pues al no admitir repetidos, no es posible unir un montículo consigo mismo. Esta operación tiene coste logarítmico.

```
void unionwith(int const& h1, int const& h2) {
        if (h1 <= 0 || h1 > heaps.size())
                throw domain_error("Heap 1 does not exist");
        else if (h1 <= 0 || h1 > heaps.size())
                throw domain_error("Heap 2 does not exist");
        else if (h1 == h2)
                throw domain_error("Heap 1 is equal to heap 2");
        heaps[h1]->unionWiht(*heaps[h2]);
        heaps[h2] = new binomialHeap<T>();
}
```

Luego se llama a la operación unionWith() de la clase montículo sobre uno de los dos montículos, y el otro se vacía creando un nuevo montículo sin elementos sobre él.

```
void unionWith(binomialHeap const& H) {
        privateUnion(H);
        nelems += H.nelems;
}
```

Esta función simplemente llama a la ya famosa función privada de la clase montículo, privateUnion(), que a continuación veremos con detalle. Además, actualiza el número de elementos.

Lo primero que hacemos en privateUnion() es comprobar que el puntero al primer elemento del montículo con el que queremos unir el nuestro no sea nulo. Si el nuestro es nulo nos quedamos con el que nos pasan, y ya habríamos terminado. En caso contrario, llamamos a la función auxiliar merge(), que fusiona las dos listas de raíces manipulando punteros (para más detalle consultar el .h), y a continuación ejecutamos el siguiente bucle:

```
Link prevx = nullptr;
Link x = head;
Link nextx = head->sibling;
while (nextx != nullptr) {
        if (x->degree != nextx->degree ||
                nextx->sibling != nullptr &&
                nextx->sibling->degree == x->degree) {
                prevx = x;  x = nextx;
        }
        else if (x->key <= nextx->key) {
                x->sibling = nextx->sibling;
                binomialLink(nextx, x);
        }
        else if (prevx == nullptr) {
                head = nextx;
                binomialLink(x, nextx);
                x = nextx;
        }
        else {
                prevx->sibling = nextx;
                binomialLink(x, nextx);
                x = nextx;
        }
        nextx = x->sibling;
}
```

Puede observarse que se llama repetidamente a la función binomialLink(), que convierte al primer elemento en el encabezado de la lista de hijos del segundo elemento.
Por último se actualiza el mínimo del montículo.

4.6. DecreaseKey(T o, T n)

Para decrecer la clave manteniendo el coste asociado a un montículo binomial, es necesario llevar el diccionario del que ya hemos hablado con anterioridad, donde se guardan los elementos de la familia de montículos. Por tanto, lo primero que se hace es comprobar que la clave que se quiere decrementar existe, y la nueva clave no existe (pues no admitimos repetidos, como bien recordaréis). Una vez comprobado todo esto, se llama a la función decreaseKey() de la clase montículo. Esta operación tiene coste logarítmico.

```
void decreaseKey(T const& o, T const& n) {
        auto it = elements.find(o);
        if (it == elements.end())
                throw domain_error("Key not foud");
        auto it2 = elements.find(n);
        if (it2 != elements.end())
                throw domain_error("Key already in the heap family");
        try {
                heaps[elements[o].second]->
                    decreaseKey(elements[o].first, n);
                auto old = elements[o];
                elements.erase(o);
                elements.insert({ n, old });
        }
        catch (domain_error e){
                throw;
        }
}
```

Lo primero que hacemos en la función decreaseKey() de la clase montículo es comprobar que la nueva clave es, en efecto, menor que la anterior. Después hacemos el intercambio de punteros necesario para decrecer la clave. Finalmente recalculamos el mínimo por si hubiera cambiado al decrecer la clave.

```
void decreaseKey(Link const& x, T const& k) {
        if (k > x->key)
                throw domain_error("New key is greater
                    than current key");
        x->key = k;
        Link y = x;
        Link z = y->parent;
        while (z != nullptr && y->key < z->key) {
                T aux = y->key;
                y->key = y->parent->key;
                y->parent->key = aux;

                y = z;
                z = y->parent;
        }
        if (min == nullptr || k < min->key)
                min = x;
        if (head == nullptr)
                head = x;
}
```

4.7. DeleteKey(T e)

A la hora de querer eliminar un elemento, la lógica nos dice que lo primero que debemos hacer es comprobar que, en efecto, el elemento que queremos eliminar existe. Así pues lo hacemos en esta función, echando mano del diccionario donde tenemos todos los elementos almacenados. Una vez comprobado que realmente existe, nos disponemos a eliminar dicho elemento. Esta operación tiene coste logarítmico.

```
void deleteKey(T const& e) {
        auto it = elements.find(e);
        if (it == elements.end())
                throw domain_error("Unvalid key");
        if (e == heaps[elements[e].second]->min->key)
                heaps[elements[e].second]->extractMin();
        else {
                Link aux = heaps[elements[e].second]->
                    deleteKey(elements[e].first);
                if (elements.size() > 1){
                        elements[heaps[elements[e].second]->
                          minimum()] = { aux, elements[e].second };
               }
        }
        elements.erase(e);
        totalElems--;
```

Para ello llamamos a la función deleteKey() de la clase montículo. Lo que realiza esta función es algo muy sencillo una vez implementada la función decreaseKey(), pues nos basta con decrecer la clave del elemento que queremos eliminar al mínimo posible, guardándonos antes el mínimo en una variable auxiliar y eliminándolo (pues, como sabemos, nuestros montículos no admiten repetidos). Así, nuestro elemento se habría convertido en el nuevo mínimo, por lo que solo necesitaríamos extraer el mínimo para eliminarlo. Por último, volvemos a insertar nuestro mínimo que teníamos guardado en la variable auxiliar.

```
Link deleteKey(Link const& x) {
        T aux = min->key;
        Link minimum = min;
        if (head == nullptr)
                head = min;
        extractMin();
        decreaseKey(x, aux);
        extractMin();
        Link h = insert(aux);
        if (head == minimum)
                head = h;
        return h;
}
```

4.8. Size(int n)

Devuelve el tamaño del montículo n utilizando la función size() del montículo que devuelve el atributo que lleva el número de elementos de dicho montículo. Esta operación tiene coste constante.

```
size_t size(int const& n) const{
        if (n <= 0 || n > heaps.size())
                throw domain_error("This heap does not exist");
        return heaps[n]->size();
}
```

4.9. GlobalSize()

Devuelve el tamaño de la familia de montículos utilizando el atributo de la familia de montículos que lleva el número de elementos totales. Esta operación tiene coste constante.

```
size_t globalSize() const{
        return totalElems;
}
```

4.10. Empty()

Comprueba si la familia de montículos está vacía utilizando para ello el número de elementos de la misma. Esta operación tiene coste constante.

```
bool empty() const{
        return totalElems == 0;
}
```

4.11. IsElem()

Comprueba si el elemento en cuestión se encuentra en la familia de montículos buscándolo en el diccionario donde llevamos todos los elementos de la familia. Esta operación tiene coste constante.

```
bool isElem(T const& e) {
        return elements.count(e);
}
```

4.12. EmptyKey()

Comprueba si el montículo en cuestión existe verificando que el puntero al mínimo no es nulo. Esta operación tiene coste constante.

```
bool emptyKey(int const& n) {
        if (n <= 0 || n > heaps.size())
                throw domain_error("This heap does not exist");
        return heaps[n]->min == nullptr;
}
```

Capítulo 5

Pruebas

5.1. Pruebas de métodos

Las pruebas de los distintos métodos se basan todas en introducir, de forma aleatoria, un número elevado de elementos en nuestra familia de montículos, y hacerlo a la vez en la PriorityQueue proporcionada en clase, y comprobar, tras realizar una serie de operaciones, que ambos montículos contienen los mismos elementos.
Esta es una forma muy fiable de probar nuestra implementación, pues al insertar un número muy elevado de elementos de forma aleatoria, y comprobar que nuestra familia de montículos se comporta de manera similar a la clase PriorityQueue, estamos realizando un testeo muy grande y, por tanto, completo (gracias, sobre todo, a la aleatoriedad de los elementos insertados y a la corrección de la clase PriorityQueue proporcionada en clase). [6] [7]

5.1.1. Prueba de insert() y minimum()

En esta prueba se testean el método de inserción en un montículo concreto y el método de mostrar el mínimo de un montículo de la forma explicada anteriormente.

```
TEST_METHOD(insertAndMinimum){
        //srand(time(NULL));
        srand(3000);
        PriorityQueue <int> queue;
        binomialHeapFamily <int> binheap;
        int random = rand() % 300000;
        binheap.insert(random);
        queue.push(random);

        for (int i = 1; i < 200000; i++) {
                int random = rand() % 300000;
                queue.push(random);
                binheap.insert(random, 1);
                int min = queue.top();
                int min2 = binheap.minimum(1);
                Assert::AreEqual(min, min2);
        }
}
```

5.1.2. Prueba de extractMin()

En esta prueba se testea el método de extractMin() insertando una serie de elementos y extrayendo el mínimo uno a uno.

```
TEST_METHOD(extractMin) {
        srand(3000);
        PriorityQueue <int> queue;
        binomialHeapFamily <int> binheap;
        int random = rand() % 300000;
        binheap.insert(random);
        queue.push(random);

        for (int i = 1; i < 200000; i++) {
                int aleatorio = rand() % 300000;
                if (!binheap.isElem(random)) {
                        queue.push(random);
                        binheap.insert(random, 1);
                        Assert::AreEqual(queue.top(),
                            binheap.minimum(1));
                        queue.pop();
                        binheap.extractMin(1);
                }
        }
}
```

5.1.3. Prueba de decreaseKey()

En esta prueba se testea el método decreaseKey(), generando tanto los elementos insertados como el que queremos decrecer y al que queremos decrecer, de forma aleatoria (para lo que utilizamos srand).

```
TEST_METHOD(decreaseKey) {
        srand(3000);
        IndexPQ <int, less<int>> queue(200000);
        binomialHeapFamily <int> binheap;
        int random = rand() % 300000;
        binheap.insert(random);
        queue.push(random, random);

        for (int i = 0; i < 200000; ++i) {
                int random = rand() % 300000;
                if (!binheap.isElem(random)) {
                        queue.push(random, random);
                        binheap.insert(random, 1);
                }
        }
        for (int j = 0; j < 200000; j++) {
                int bigRandom = rand() % 300000;
                int smallRandom = rand() % 30000;

                if (binheap.isElem(bigRandom) &&
                    !binheap.isElem(smallRandom) &&
                        smallRandom < bigRandom) {
                            queue.update(bigRandom, smallRandom);
                            binheap.decreaseKey(bigRandom,
                                smallRandom);
                            int min = queue.top().prioridad;
                            int min2 = binheap.minimum(1);
                            Assert::AreEqual(min, min2);
                }
        }
}
```

5.1.4. Prueba de deleteKey()

En esta prueba se testea el método deleteKey() insertando una serie de elementos de forma aleatoria y borrándolos uno a uno mientras el montículo no este vacío.

```
TEST_METHOD(deleteKey) {
        srand(3000);
        binomialHeapFamily <int> binheap;
        vector <int> v;
        int random = rand() % 300000;
        binheap.insert(random);
        v.push_back(random);


        for (int i = 0; i < 200000; i++) {
                int aleatorio = rand() % 300000;
                if (!binheap.isElem(random))
                        v.push_back(random);
                binheap.insert(random, 1);
        }
        while (!v.empty()) {
                binheap.deleteKey(v.back());
                v.pop_back();
        }

        Assert::AreEqual(true, binheap.emptyKey(1));
}
```

5.2. Pruebas de excepciones

En esta sección comprobamos que, en efecto, las excepciones controladas en el código son lanzadas y recogidas, y que funcionan a la hora de evitar errores. Solo se adjuntan algunas pruebas de excepciones, ya que son muchas y todas muy parecidas. Para más información consultar el .h.

5.2.1. minNotFound

En esta prueba comprobamos que se lanza la excepción cuando intentamos acceder al mínimo de un montículo vacío.

```
TEST_METHOD(minNotFound){
        binomialHeapFamily<int> binheap;
        try {
                binheap.minimum(1);
                Assert::Fail();
        }
        catch (domain_error d) {
        }
}
```

5.2.2. keyAlreadyExists

En esta prueba comprobamos que se lanza la excepción cuando intentamos decrecer una clave a otra que ya pertenece a la familia de montículos.

```
TEST_METHOD(keyAlreadyExists){
        binomialHeapFamily<int> binheap;
        binheap.insert(1);
        binheap.insert(2);
        try {
                binheap.decreaseKey(2, 1);
                Assert::Fail();
        }
        catch (domain_error d) {
        }
}
```

5.2.3. nonExistingHeap

En esta prueba comprobamos que se lanza la excepción cuando intentamos insertar un elemento en un montículo que no existe en la familia.

```
TEST_METHOD(nonExistingHeap){
        binomialHeapFamily<int> binheap;
        try {
                binheap.insert(3, 5);
                Assert::Fail();
        }
        catch (domain_error d) {
        }
}
```

Bibliografía

[1] Black, Paul E. y Vreda Pieterse: *Dictionary of Algorithms and Data Structures*. National Institute of Standards and Technology, 1998.

[2] Brassard, Gilles y Paul Bratley: *Fundamentos de Algoritmia*. Prentice Hall, 1998.

[3] Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest y Clifford Stein: *Introduction to Algorithms*. McGraw-Hill Book Company, Cambridge, Massachusetts London, England, segunda edición, 1990.

[4] Horowitz, Ellis, Dinesh Mehta y Sartaj Sahni: *Fundamentals of Data Structure in C++*. W. H. Freeman Co, New York, NY, USA, 1995.

[5] Oliet, Narciso Martí, Yolanda Ortega Mallén y Alberto Verdejo: *Estructuras de Datos y Métodos Algorítmicos*. Ibergaceta Publicaciones, SL, Madrid, segunda edición, 2013.

[6] Verdejo, Alberto: *IndexPQ.h*.

[7] Verdejo, Alberto: *PriorityQueue.h*.

[8] Vuillemin, Jean: *A Data Structure for Manipulating Priority Queues*. S.L. Graham, R.L. Rivest Editors, Université de Paris-Sud, 1978.