Universitat Politècnica de Catalunya

Facultat d'Informàtica de Barcelona

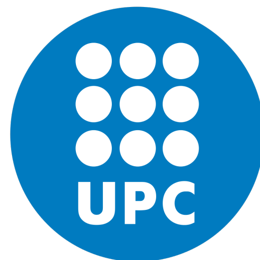
Grau en Enginyeria Informàtica - Computació

Randomized Quad Trees: Implementation and Experimental Analysis

Autor:
Oliver Martínez Pons

Directora:
Amalia Duch Brown

3 de Julio de 2019



Resumen

Los quad trees propuestos por Bentley en el 1974 [7] son una estructura de datos diseñada para resolver búsquedas asociativas [8]. Esta estructura es una generalización de los árboles binarios de búsqueda y es interesante porque es clásica y es utilizada. Además, es una estructura de datos jerárquica de propósito general que puede ser generalizada fácilmente a múltiples dimensiones [14].

El problema de los quad trees es que es difícil borrar elementos [13] o, dicho de otra manera, que sean dinámicos. Pese a que el borrado en dos dimensiones está definido por H. Samet [13], el algoritmo es complicado y difícil de implementar. Es más, este algoritmo también es difícil de generalizar para dimensiones mayores a 2.

Además, en el algoritmo clásico de inserción, la forma del árbol depende del orden en el que se insertan las llaves. Si las llaves son generadas de manera independiente por una distribución de probabilidad continua, se obtiene un árbol aleatorio [5] cuya altura esperada es logarítmica con respecto al número de llaves del árbol, sin embargo, si las llaves se insertan en orden, la altura del árbol es lineal, lo que tiene un impacto en la eficiencia de las búsquedas exactas en dicho árbol (que en el caso peor son de coste proporcional a la altura del árbol) y también en el de las búsquedas asociativas.

En el artículo, *Randomized insertion and deletion in point quad trees* [4], se propone, utilizando algoritmos aleatorios [10], un algoritmo de borrado simple y escalable a más dimensiones. El algoritmo es fácil de describir y de generalizar y es bastante más sencillo de implementar que el de Samet. Además, los algoritmos de inserción y borrado en randomized quad trees garantizan que los quad trees resultantes sean siempre aleatorios.

En este trabajo final de grado se estudian los algoritmos randomizados propuestos en el artículo citado anteriormente, se implementan y se propone una implementación alternativa y más eficiente que la original. Finalmente, se analiza experimental y exhaustivamente la eficiencia de los algoritmos de inserción y borrado propuestos.

Posteriormente se comparan los resultados con algoritmos alternativos; una implementación con cola de prioridad y una implementación sin randomizar.

Los resultados experimentales muestran que los algoritmos randomizados propuestos: 1) funcionan correctamente para cualquier dimensión (de hecho se han probado hasta dimensión 6, pero los al-

goritmos son válidos para cualquier dimensión, especialmente el borrado), 2) producen árboles aleatorios que cumplen con los costes esperados dados en la literatura [4], 3) compiten en eficiencia con la implementación con cola de prioridad propuesta especialmente para dimensiones menores de 5, y 4) compiten en eficiencia (son mucho mejores) que el algoritmo de inserción clásico en el caso en el que las llaves a insertar no sean aleatorias o estén dadas en orden. De hecho, este último algoritmo no produce árboles aleatorios cuando las llaves no son generadas de manera aleatoria [3, 2].

Resum

Els quad trees proposats per Bentley en el 1974 [7] són una estructura de dades dissenyada per resoldre cerques associatives [8]. Aquesta estructura és una generalització dels arbres binaris de cerca i són interessants perquè és clàssica i és emprada. A més a més, és una estructura de dades jeràrquica de propòsit general que pot ser generalitzada fàcilment per a múltiples dimensions [14].

El problema dels quad trees és que és difícil esborrar elements [13] o, dit d'altra manera, que siguin dinàmics. Malgrat que l'esborrat en dues dimensions està ben definit per H. Samet [13], l'algoritme és complicat i difícil d'implementar. És més, aquest algoritme també és difícil de generalitzar per a dimensions majors a 2.

A més a més, en l'algoritme clàssic d'inserció, la forma de l'arbre depèn de l'ordre en què s'insereixen les claus. Si les claus són generades de manera independent per una distribució de probabilitat contínua, s'obté un arbre aleatori [5] el qual l'altura esperada és logarítmica respecte el nombre de claus del arbre, tanmateix, si les claus s'insereixen en ordre, l'altura de l'arbre és lineal, cosa que té un impacte en l'eficiència de les cerques exactes en aquest arbre (que en el pitjor cas són de cost proporcional a l'altura de l'arbre) i també en el de les cerques associatives.

En l'article, *Randomized insertion and deletion in point quad trees* [4], es proposa, emprant algoritmes aleatoris [10], un algoritme de esborrat simple i escalable a més dimensions. L'algoritme és fàcil de descriure i de generalitzar i és bastant més senzill d'implementar que el de Samet. A més a més, els algoritmes d'inserció i esborrat en randomized quad trees garanteixen que els quad trees resultants siguin sempre aleatoris.

En aquest treball final de grau s'estudien els algoritmes randomizats proposats en l'article citat anteriorment, s'implementen i es proposa una implementació alternativa i més eficient que l'original. Finalment, s'analitza experimental i exhaustivament l'eficiència dels algoritmes d'inserció i esborrat proposats.

Posteriorment es comparen els resultats amb algoritmes alternatius; una implementació amb cua de prioritat i una implementació sense randomitzar.

Els resultats experimentals mostren que els algoritmes randomizats proposats: 1) funcionen correctament per a qualsevol dimensió (de fet s'han provat fins a dimensió 6, però els algoritmes són vàlids per a qualsevol dimensió, especialment el esborrat), 2) produeixen arbres

aleatoris que compleixen amb els costos esperats donats en la literatura [4], 3) competeixen en eficiència amb la implementació amb cua de prioritat proposta especialment per a dimensions menors que 5, i 4) competeixen en eficiència (són molt millors) que l'algoritme d'inserció clàssic en el cas en què les claus a insertar no siguin aleatòries o estén donades en ordre. De fet, aquest últim algoritme no produeix arbres aleatoris quan les claus no són generades de manera aleatòria [3, 2].

Abstract

The quad trees proposed by Bentley in 1974 [7] are a data structure designed to solve associative queries [8]. This structure is a generalization of binary search trees and is interesting because it is classical and it is used. In addition, it is a general-purpose hierarchical data structure that can easily be generalized to multiple dimensions [14].

The problem with quad trees is that it is difficult to delete elements [13], or in other words, to be dynamic. Although the two-dimensional deletion is defined by H. Samet [13], the algorithm is complicated and difficult to implement. Moreover, this algorithm is also difficult to generalize for dimensions greater than 2.

In addition, in the classic algorithm of insertion, the shape of the tree depends on the order in which the keys are inserted, if the keys are generated independently by a continuous probability distribution, a random tree is obtained [5], whose expected height is logarithmic in regard to the number of keys of the tree, however, if the keys are inserted in order, the height of the tree is linear, which has an impact on the efficiency of the exact searches in said tree (which in worst case their cost is proportional to the height of the tree) and also in the associative queries.

In the article, *Randomized insertion and deletion in point quad trees* [4], it is proposed, using random algorithms [10], a simple and scalable deletion algorithm for more dimensions. The algorithm is easy to describe and generalize and is much simpler to implement than the Samet algorithm. In addition, the insertion and deletion algorithms in randomized quad trees guarantee that the resulting quad trees are always random.

In this final degree project, the randomized algorithms proposed in the aforementioned article are studied and implemented, and an alternative and more efficient implementation than the original one is proposed. Finally, the efficiency of the proposed insertion and deletion algorithms is experimentally and exhaustively analysed.

Subsequently, the results are compared with alternative algorithms; an implementation with priority queue and an implementation without randomization.

The experimental results show that the proposed randomized algorithms: 1) work correctly for any dimension (in fact they have been tested up to dimension 6, but the algorithms are valid for any dimension, especially the deletion), 2) produce random trees that meet

the costs expected in the literature [4], 3) compete in efficiency with the implementation with priority queue proposed especially for dimensions less than 5, and 4) compete in efficiency (are much better) than the classical insertion algorithm in the case where the keys to be inserted are not random or are given in order. In fact, this last algorithm does not produce random trees when the keys are not generated randomly [3, 2].

Contenidos

1 Introducción 9

2 Preliminares 11

2.1 Búsquedas asociativas 11

2.2 Definición de Quad trees 12

3 Randomized Quad trees 17

3.1 Definición 17

3.2 Implementación original 18

4 Algoritmos e Implementación 20

5 Experimentación y análisis experimental 25

5.1 Costes de los algoritmos 25

5.1.1 Coste de creación 26

5.1.2 Coste de insertado 28

5.1.3 Coste de borrado 30

5.1.4 Coste de la inserción en la raíz 32

5.1.5 Coste del borrado en la raíz 34

5.1.6 Análisis de los costes 36

5.2 Comparativas 37

5.2.1 Coste de creación 38

5.2.2 Coste de inserción 41
5.2.3 Coste de borrado 43

6 Planificación y análisis económico **46**

6.1 Planificación 46
6.2 Análisis económico 48

7 Sostenibilidad **50**

7.1 Dimensión ambiental 50
7.2 Dimensión económica 50
7.3 Dimensión social 51

8 Conclusiones y trabajo futuro **52**

8.1 Conclusiones 52
8.2 Trabajo futuro 53

1 Introducción

Un problema computacional común son las búsquedas asociativas. Este problema se presenta, por ejemplo, cuando utilizamos los teléfonos móviles y queremos saber cuál es el restaurante más cercano o las gasolineras que se encuentran dentro de una determinada región. Estas búsquedas son relevantes en distintos campos como pueden ser los sistemas de información geográfica, las bases de datos o los gráficos por computador y son utilizada por aplicaciones como Google Maps y Tripadvisor. Por ejemplo, digamos que tenemos una serie de localizaciones de restaurantes en Barcelona y sus coordenadas. Estas búsquedas nos permitirían buscar eficientemente los restaurantes que se encuentren en un cierto rango o cual es el restaurante más cercano respecto a nuestra posición actual.

Para resolver el problema de las búsquedas asociativas en el 1974 se introducen los quad trees con la publicación del artículo *Quad trees: a data structure for retrieval on composite key* [7]. En este artículo se mostraba una estructura de datos que consistía en generalizar los árboles binarios de búsqueda para múltiples dimensiones, los quad trees. A partir de ese momento se han presentado muchas más estructuras de datos multidimensionales [14, 15]. Los quad trees no son las únicas que se utilizan; existen diversas estructuras y cada una de ellas tiene sus ventajas e inconvenientes. Cada una de ellas es utilizada en concordancia con el contexto de la aplicación para la que van a ser utilizadas.

R. A. Finkel and J. L. Bentley [7] vieron que en los quad trees bien balanceados el insertado seguía siendo de coste logarítmico en el tamaño del árbol como ya pasaba en los árboles binarios de búsqueda [1]. Las búsquedas en el árbol eran también eficientes. Propusieron un algoritmo para optimizar los árboles y garantizaron que las búsquedas eran eficientes en estos. Desde entonces se estudiaron mucho todas sus propiedades y los costes para realizar los distintos tipos de búsquedas asociativas [13]. Pese a esto, el borrado seguía siendo una operación difícil.

Más tarde, H. Samet en *Deletion in two-dimensional quad-trees* [13] definió un algoritmo para el borrado en dos dimensiones que trataba el problema de una manera similar al borrado en arboles binarios de búsqueda. Comparó

su algoritmo con tener que reinsertar todos los nodos en los subárboles del nodo eliminado. Sus análisis mostraron que el número de nodos reinsertados disminuía considerablemente. Esto mejoraba la longitud total del camino del árbol y resultaba en un árbol más balanceado. El problema es que el algoritmo se vuelve demasiado complejo para dimensiones mayores de 2 [13].

En 2004 en *Randomized insertion and deletion in point quad trees* [4], se introdujeron algoritmos randomizados [12] para la inserción y el borrado en quad trees. Estos algoritmos son simples y están definidos para cualquier dimensión $k \geq 2$.

La propuesta mencionada anteriormente es teórica y sería conveniente analizar experimentalmente que resultados da en comparación a los algoritmos clásicos de borrado [13]. Y justamente ese es el objetivo de nuestro proyecto, hacer una implementación de la propuesta de los quad trees randomizados [4] para poder hacer un estudio experimental exhaustivo [11] además de dejarla disponible vía open source a la comunidad.

En estos experimentos se investigará si el algoritmo de borrado propuesto en este trabajo es más eficiente que la reconstrucción del quad tree. También se investigará cómo evoluciona el algoritmo al escalar dimensiones. Esto significa medir como varía el coste dependiendo de la dimensión; probar si deja de ser eficiente a partir de una dimensión concreta y estimarla.

Para ello, primeramente, explicaremos en el Capítulo 2 en qué consisten las búsquedas asociativas y los quad trees. Más adelante en el Capítulo 3 definiremos la propuesta original de randomized quad trees y veremos un ejemplo de ejecución de la implementación realizada. Seguidamente en el Capítulo 4 veremos en detalle la implementación realizada. Una vez hecho esto, en el Capítulo 5 veremos los resultados de los experimentos y analizaremos los costes de los algoritmos. Comparando también el coste con las otras dos alternativas implementadas. En el Capítulo 6 detallaremos la planificación del proyecto y realizaremos un análisis económico. Acabando, en el Capítulo 7 veremos la sostenibilidad del proyecto y, por último, en el Capítulo 8 resumiremos las conclusiones.

2 Preliminares

2.1 Búsquedas asociativas

Definiremos el problema de las búsquedas asociativas de la siguiente manera. Dada una colección de tuplas de k valores (ya sean coordenadas o atributos) pertenecientes a un dominio ordenado, consideraremos una búsqueda el hecho de buscar las tuplas que cumplan ciertas condiciones sobre sus atributos. La búsqueda se considerará asociativa si especifica condiciones sobre más de un atributo [8].

Un poco más formalmente, en el resto del documento, vamos a considerar que las llaves son puntos en un espacio k-dimensional tales que dada una llave x, $x = (x_0, x_1, \ldots, x_{k-1})$. También vamos a considerar que cada x_i pertenece a un dominio $D_i = [0, 1]$ y que por tanto el universo de las llaves es el hipercubo $D = [0, 1]^k$.

Dada una colección $\mathcal{F}$, de n registros (o llaves), donde cada llave de $\mathcal{F}$ es una k-tupla de valores, (los atributos o las coordenadas de la llave) provenientes de un dominio totalmente ordenado, una consulta (o búsqueda) por las llaves de $\mathcal{F}$ es un conjunto de condiciones que deben ser satisfechas por los atributos de las llaves de $\mathcal{F}$.

Como ya comentamos, la consulta se considera asociativa si solo si especifica condiciones relativas a más de uno de los atributos de las llaves.

En el resto de documento usaremos la notación que acabamos de mencionar para referirnos al conjunto de puntos, dominios, tuplas, etc...

Ejemplos de búsquedas asociativas:

- Búsquedas exactas: Estas búsquedas son las más sencillas y consisten en buscar una llave con la que coincidan todos los atributos.
- Búsquedas parciales: Las búsquedas parciales especifican un grupo de atributos que se quieren buscar y su valor. El objetivo es encontrar las llaves cuyos atributos especificados sean iguales.

- Búsquedas por región: En estas búsquedas se determina una región en el espacio de búsqueda y el objetivo es encontrar todas las llaves que estén dentro de esta.

Por ejemplo, dado un conjunto de llaves bidimensionales $\mathcal{F}$= [(1,4), (6,3), (4,2), (9,8), (5,4)], una búsqueda exacta podría ser buscar la llave (4,2), una búsqueda parcial podría ser buscar las llaves cuyo segundo atributo sea igual a 4 y una búsqueda por región podría ser buscar las llaves con el primer y segundo atributo entre 3 y 7.

Hay muchas estructuras de datos para atender estas consultas entre ellas los quad trees [7]. Estos son interesantes porque son una estructura de datos clásica y además utilizada. Estos usos van desde el tratamiento de imágenes o los sistemas de información geográficos hasta las bases de datos grandes [13]. Son una estructura de datos jerárquica y son de propósito general en el sentido de que sirven para responder a muchas consultas asociativas distintas de manera razonablemente eficiente.

2.2 Definición de Quad trees

A grandes rasgos, los quad trees de dos dimensiones consisten en un árbol cuaternario donde cada nodo tiene una llave de dos dimensiones y tiene asociado cuatro subárboles que corresponden a los cuadrantes Noroeste, Noreste, Sudeste y Sudoeste. Si consideramos que la raíz de cada subárbol parte el espacio bidimensional en cuatro cuadrantes respecto a sus coordenadas, entonces cada nuevo punto insertado que pertenezca a un cuadrante será guardado en su respectivo subárbol. Para determinar a qué cuadrante pertenece un hijo se comprobarán las coordenadas respecto a su padre. El árbol se va dividiendo recursivamente en cuadrantes hasta que no quedan puntos que guardar.

La definición formal es la siguiente:

Definición 1 *Un quad tree para una colección de llaves 2-dimensionales, $F = \{x^{(1)}, x^{(2)}, \ldots, x^{(n)}\}$, es un árbol cuaternario en el cual:*

1. *Cada nodo de F contiene una llave 2-dimensional y tiene asociado*

cuatro subárboles correspondientes a los cuadrantes NO, NE, SE y SO.

2. *Para cada nodo con llave x la siguiente invariante se cumple: cada llave y en el subárbol NO satisface que $y_1 < x_1$ y $y_2 \geq x_2$; cada llave y en el subárbol NE satisface que $y_1 \geq x_1$ y $y_2 \geq x_2$; cada llave y en el subárbol SE satisface que $y_1 \geq x_1$ y $y_2 < x_2$; y, cada llave y en el subárbol SO satisface que $y_1 < x_1$ y $y_2 < x_2$.*

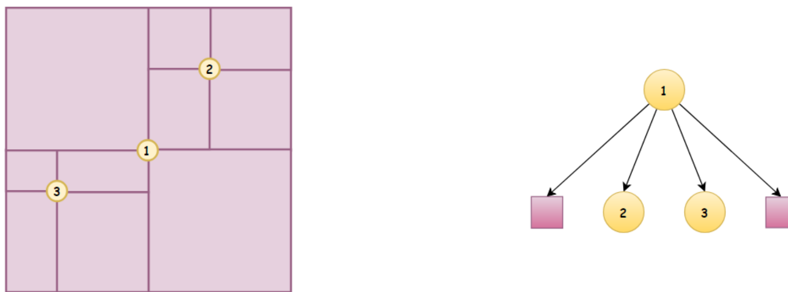


Figura 1: A la izquierda un quad tree representado en un plano y a la derecha en un árbol.

La búsqueda y la inserción de llaves son muy similares entre sí. Si queremos buscar un registro, tenemos que tomar la raíz y comparar cada coordenada de la llave que se insertará con cada coordenada de la raíz. Si el nodo no se encuentra en la raíz, avanzaremos en la dirección correcta y haremos esto recursivamente hasta que encontremos el punto o haya un nodo vacío. En este último caso, la llave buscada no se encuentra en el árbol.

Si queremos insertar un registro, tenemos que ir a través del árbol de la misma manera que en la búsqueda hasta que encontremos un nodo vacío. En este caso podremos insertar el registro que a su vez dividirá el subcuadrante en 4 particiones más.

El algoritmo de borrado clásico consiste en buscar la llave en el árbol y, si se encuentra, eliminar el nodo y reinsertar todas las llaves que pertenecían al subcuadrante en la raíz del árbol inicial. Podemos ver que este algoritmo no es el más eficiente, pero si el más simple. El orden en el que los puntos se

insertan en el árbol es muy importante. Un mal orden puede resultar en un quad tree desequilibrado.

Veamos las operaciones con un ejemplo. Partiendo de un espacio en un plano cartesiano que representa los posibles valores de las llaves dentro del dominio de los números reales, denotaremos los cuadrantes, generados al partir el espacio en un punto, Noroeste, Noreste, Sudeste y Sudoeste; NO, NE, SE y SO respectivamente. Y supondremos que hacia arriba las y incrementan y hacia la derecha las x incrementan.

Partiremos del árbol de la Figura 2 y vamos a insertar la llave *4*. Como la raíz ya está ocupada el algoritmo compara en que cuadrante el punto debe ser colocado respecto al nodo *1*. En este caso la llave tiene que ir en el primer cuadrante porque su coordenada x es menor que la de *1* y su coordenada y es mayor. El algoritmo avanza a ese subárbol y como está vacío el nodo es insertado.

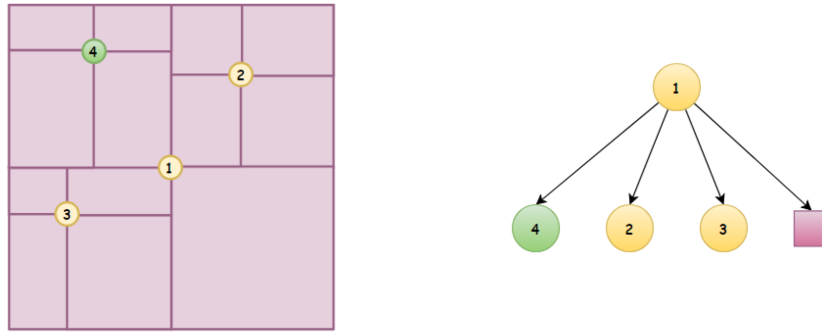


Figura 2: Inserción de un nodo 4 en el quad tree.

Ahora vamos a insertar la llave *5*. En este caso volvemos a aplicar el algoritmo, pero esta vez, hace falta que avancemos dos niveles hacia abajo ya que este se encuentra al noroeste del primer nodo y por tanto tiene que bajar un primer nivel y posteriormente tiene que descender al cuarto cuadrante del nodo 4, ya se encuentra al sudeste de este, para colocarse en las hojas del árbol.

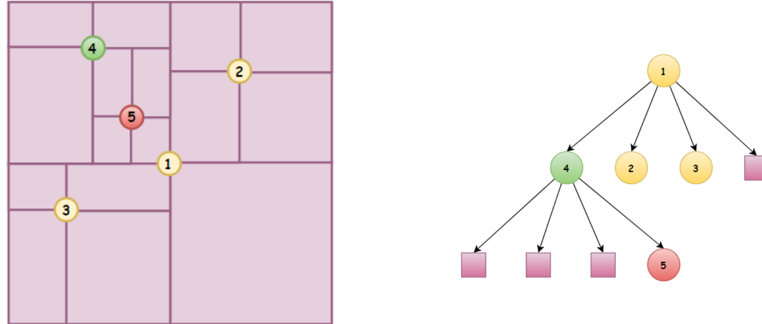


Figura 3: Inserción de un nodo 5 en el quad tree.

Ahora vamos a borrar la llave *4*. El algoritmo la busca y la elimina. Ahora tenemos que hacer algo con *5*. Como hemos descrito anteriormente, el algoritmo clásico reinsertara el nodo *5* desde la raíz siguiendo el algoritmo de insertado quedando de la siguiente manera:

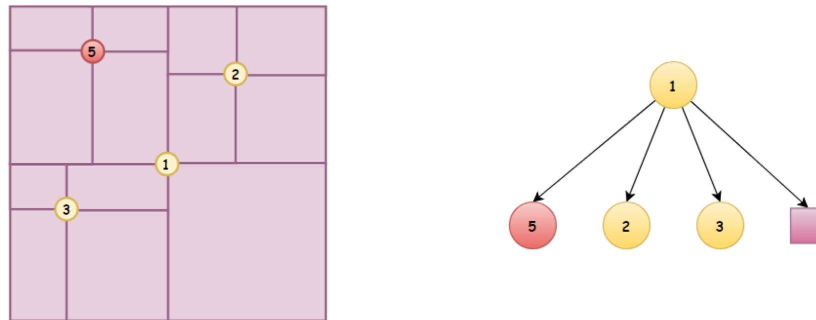


Figura 4: Quad tree tras borrar el nodo 4.

Los quad trees pueden ser generalizados fácilmente a múltiples dimensiones [14]. Un quad tree k-dimensional consiste pues, en un árbol k-ario donde cada nodo guarda una tupla de k valores y tiene 2^k subárboles [7]. Por tanto, cada llave del árbol dividirá el espacio de búsqueda en 2^k hipercuadrantes donde cada uno de ellos corresponderá a una permutación de comparaciones donde cada atributo es mayor o menor que el del nodo padre.

Y la definición formal es la siguiente:

Definición 2 *Un quad tree T de tamaño $n \geq 0$ es una colección de n registros k-dimensionales, que consisten en una llave $x = (x_0, \ldots, x_{k-1}) \in D$, donde $D = D_0 \times \cdots \times D_{k-1}$, y cada D_j, $0 \leq j < k$, es un dominio completamente ordenado. El quad tree T es un árbol 2^k-ario tal que*

- *o bien es vacío y $n = 0$, o*
- *su raíz guarda una llave x y tiene 2^k subárboles, cada uno asociado a una secuencia de bits de longitud k $w = w_0 w_1 \ldots w_{k-1} \in \{0,1\}^k$, y las demás $n-1$ llaves restantes están guardadas en uno de sus subárboles T_w, tal que $\forall w \in \{0,1\}^k$: T_w es un quad tree y para cada llave $y \in T_w$, se cumple que $y_j < x_j$ si $w_j = 0$ y $y_j > x_j$ si $w_j = 1$, $0 \leq j < k$.*

El problema de los quad trees es que el algoritmo de borrado es muy costoso [13] y destruye la aleatoriedad del árbol en caso de que este lo fuese. Como ya mencionamos anteriormente, existe un borrado eficiente en dos dimensiones definido por H. Samet [13] pero el algoritmo es complicado y difícil de implementar. Es más, este algoritmo también es difícil de generalizar para dimensiones mayores que 2 y es tan complicado que la solución que se ha adoptado en la práctica es la de reconstruir el subárbol completo afectado por un borrado, lo que claramente incurre en costes más altos de los algoritmos. Otro problema del borrado es que, si tenemos un árbol generado con nodos aleatorios, tras cada borrado el árbol se desbalancea y pierde eficiencia. Es más, si las llaves que se insertan no son generadas aleatoriamente (por ejemplo, están ordenadas creciente o decrecientemente) el árbol no es aleatorio y no se cumplen los costes logarítmicos. Para resolver estos problemas podemos hacer uso de los Randomized quad trees que explicamos en la siguiente sección. Antes de continuar damos la definición formal de random quad tree que es la siguiente:

Definición 3 *Un random quad tree de tamaño n es un quad tree construido insertando n llaves independientes escogidas de una distribución de probabilidad continua definida sobre $[0,1]^k$.*

3 Randomized Quad trees

3.1 Definición

Los randomized quad trees se diferencian de los quad trees originales porque ahora los nodos además de contener una tupla de k atributos, donde k es la dimensión del árbol, estos contienen un atributo extra al que llamaremos prioridad [16].

Definición 4 *Un quad tree es un randomized quad tree si para cada nodo, los hijos de este o bien son nulos o su prioridad es inferior a la del padre.*

En particular cabe observar que si las prioridades se generan de manera independiente y uniformemente a partir del intervalo [0,1], un randomized quad tree es un random quad tree y por tanto se cumplen todas las propiedades teóricas demostradas para los random quad trees [9, 16, 10, 4].

En la Figura 5 se puede ver un quad tree con prioridades que no cumple esta condición y por tanto no es un randomized quad tree mientras que el árbol a su derecha sí que lo es.

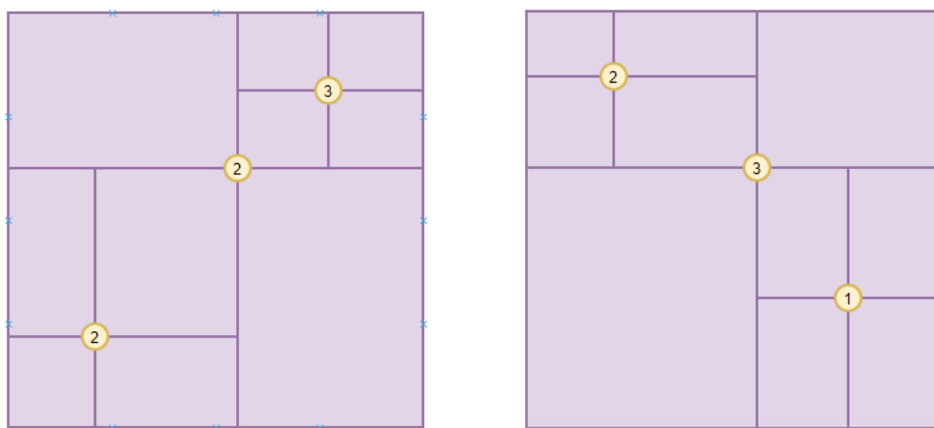


Figura 5: Quad tree no randomized a la izquierda y randomized a la derecha.

Esta propiedad que hace que los árboles sean randomized nos permite crear

árboles que no se vean afectados por el orden de los nodos insertados siempre y cuando los algoritmos de inserción y borrado nos produzcan randomized quad trees como resultado [4].

3.2 Implementación original

A continuación, vamos a ver cómo funcionan los algoritmos de inserción y borrado basados en *Randomized insertion and deletion in point quad trees* [4].

Como hemos comentado, los nodos ahora además de ser una tupla de k atributos, donde k es la dimensión del árbol, este contiene un atributo extra al que llamaremos prioridad y será un número generado uniformemente en el intervalo [0,1] y de manera independiente para cada llave. Lo que queremos con esto es que el insertado y borrado produzcan árboles que se comporten como si hubiesen sido creados por llaves generadas uniforme e independientemente.

Para ello queremos que el insertado tenga una cierta probabilidad de convertir al nuevo nodo en la raíz del árbol. De la misma manera, cuando un nodo es borrado es necesario que los nodos inferiores tengan una probabilidad de remplazarlo.

Siendo T un árbol y x un nodo a insertar, el algoritmo de inserción procederá de la siguiente manera:

1. Si el árbol T está vacío, el algoritmo crea un árbol con raíz x y 2^k subárboles vacíos.
2. Si el árbol T no está vacío y la prioridad de x es mayor que la prioridad de la raíz de T, entonces x se convertirá en la nueva raíz del árbol y sus subárboles serán el resultado de llamar a la función `split` sobre T. Si la prioridad no es mayor entonces x será insertada recursivamente en el subárbol de T que le corresponda por sus atributos.

Por otra parte, el algoritmo de borrado buscará el nodo a borrar y una vez encontrado llamará a la función `join` para juntar sus 2^k subárboles para remplazarlo como nuevo árbol.

Ahora vamos a explicar los algoritmos de `split` y `join` que utilizan los algoritmos de inserción y borrado.

Dada una llave x y un árbol T el algoritmo `split` devuelve un nuevo árbol T' con raíz x' y con los 2^k subárboles correspondientes. Es decir, para cada nodo de T, este habrá sido reasignado al subárbol correspondiente del nuevo árbol T' con raíz x'.

Por su parte el algoritmo `join` recibe un conjunto de 2^k árboles C y devuelve un nuevo árbol cuya raíz es la raíz con más prioridad de entre los 2^k árboles de C.

Los algoritmos de `split` y `join` se llaman el uno al otro para ir construyendo el árbol. El `split` parte el árbol respecto a la nueva raíz y el `join` junta los subárboles en uno solo y vuelve a llamar al `split` con la nueva raíz.

Esta propuesta fue implementada pero su coste era muy elevado y no podía competir con los demás algoritmos dado que los algoritmos de `split` y `join` recorrían el quad tree completo múltiples veces. Por ello, inspirándonos en los algoritmos propuestos, se implementó una propuesta que aprovechaba las ideas de los algoritmos de `split` y `join` y que podía competir con los demás algoritmos. Aun así, la nueva implementación podría ser mejorada aún más dado que ciertas propiedades de la implementación original que permitían ahorrar visitas a nodos no han sido incluidas aún.

4 Algoritmos e Implementación

En esta sección explicamos los algoritmos de inserción y de borrado que proponemos para los randomized quad trees. Estos funcionan igual que en la propuesta anterior. Eso sí, los algoritmos de inserción y de borrado utilizan a su vez otros dos algoritmos que aun llamarse de el mismo modo que los algoritmos originales hacen cosas distintas: el algoritmo `split` y el algoritmo `join`.

Comenzaremos presentando el algoritmo `split` cuyo código se encuentra en el Programa 2. El objetivo de este algoritmo es agrupar todos los nodos de un árbol dado según su relación de orden con una llave dada. Por tanto, este algoritmo recibe como entradas un quad tree k-dimensional T y una llave x.

En una primera etapa, este algoritmo produce una matriz de 2^k filas tal que cada fila i de la matriz contiene los elementos del árbol que irían en el subárbol i-ésimo de un nuevo randomized quad tree k-dimensional que tuviese raíz x.

En la segunda etapa para cada fila de la matriz llama al algoritmo `join` que devuelve un randomized quad tree formado por los elementos contenidos en dicha fila.

Finalmente, el algoritmo `split` devuelve un vector de randomized quad trees que contiene en cada posición i un puntero al i-ésimo subárbol del randomized quad tree con raíz x que contiene todas las llaves del árbol original T.

Cabe mencionar que el algoritmo `split` recorre el árbol T por niveles (utilizando el algoritmo clásico de recorrido en anchura de un grafo [1]).

Por su parte, el algoritmo `join` (mirar Programa 3) como ya se ha dicho, recibe como parámetro un vector de llaves y produce un randomized quad tree con esas llaves (respectando las prioridades correspondientes).

Para ello, La función `join` buscara cuál de los nodos recibidos tiene más prioridad para convertirlo en la raíz del nuevo subárbol.

Una vez seleccionado, los nodos serán vueltos a separar en una matriz como hace el algoritmo `split` pero ahora respecto al nodo que ha sido elegido como raíz del subárbol.

Por último, la función es llamada recursivamente para cada fila de la matriz una vez que los nodos han sido reasignados respecto a la nueva raíz.

Vamos a verlo con un ejemplo. Imaginemos que tenemos el quad tree bidimensional T de la Figura 6 y queremos insertar el nodo x con prioridad 9.

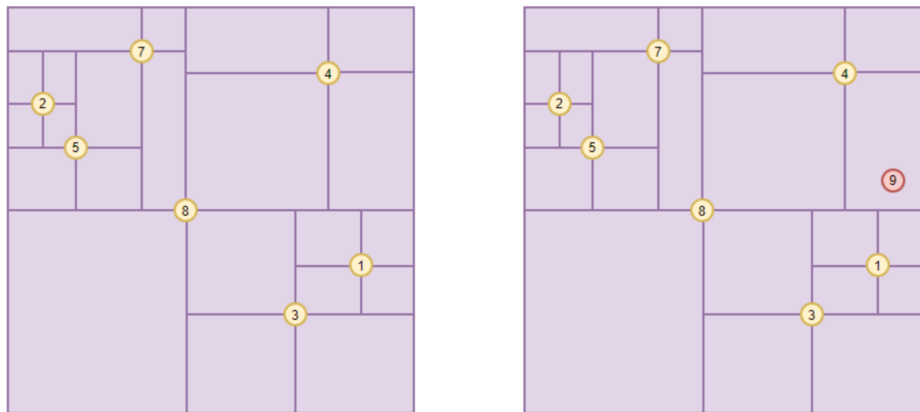


Figura 6: A la izquierda el quad tree original y a la derecha tras superponer el nodo 9 que se va a insertar.

Como el nodo tiene más prioridad que la raíz actual del árbol, se llamará a la función `split` que partirá el árbol en 4 cuadrantes respecto a 9.

Ahora se llamará a la función `join` en cada nuevo cuadrante. Esta seleccionará el nodo con más prioridad y lo hará raíz del nuevo subárbol. Ahora los nodos restantes serán vueltos a repartir respecto a la nueva raíz y serán juntados recursivamente mediante el algoritmo `join`.

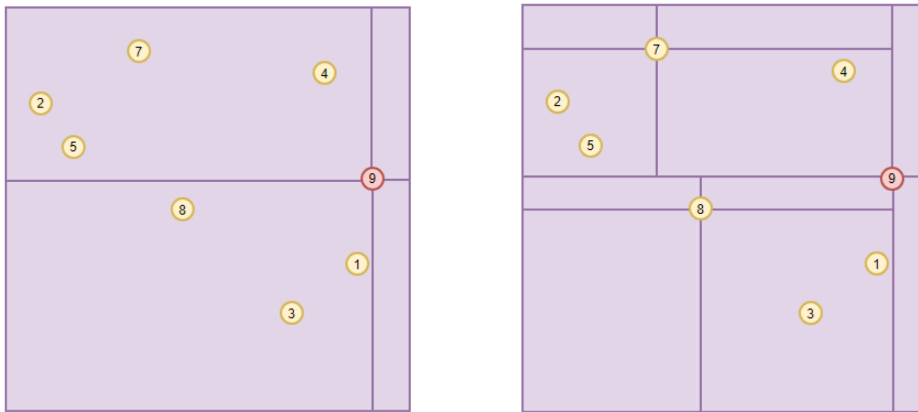


Figura 7: A la izquierda el resultado de el algoritmo `split` donde el nodo con prioridad 9 es la raíz y los demás nodos están agrupados por cuadrantes. A la derecha la ejecución del primer `join` en cada cuadrante.

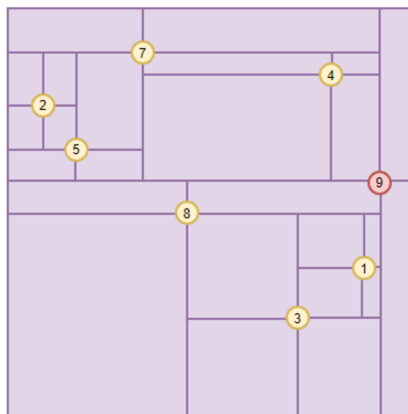


Figura 8: El resultado final tras ejecutar el algoritmo `join` recursivamente.

Antes de ver los códigos, es conveniente presentar el struct `QuadtreeNode` con el que operan los algoritmos. Cada nodo del quad tree consiste en una instancia de este struct que a su vez consiste en una llave x que contiene los k atributos seguido de la prioridad y un vector de 2^k punteros a los subárboles, que también consisten en instancias del struct `QuadtreeNode`.

Programa 1 Implementación en `C++` de la estructura `QuadtreeNode`

```
typedef vector<double> Key;

struct QuadtreeNode {
  Key x;
  vector<QuadtreeNode*> child;
  QuadtreeNode(const Key& _x) : x(_x),child(int(exp2(K)), NULL) {}
};
```

Por otra parte, tenemos la función `subtree_to_insert` que, dados dos nodos, nos devuelve un índice i que indica el i-ésimo subárbol en el que el primer nodo debería ser insertado respecto al segundo.

Programa 2 Implementación en `C++` del algoritmo `split`

```
//INPUT: una llave x y un arbol T
//OUTPUT: 2^k subarboles de x con los nodos de T
vector<QuadtreeNode*> split(const Key& x, QuadtreeNode*& T){
    vector<QuadtreeNode*> childs(int(exp2(K)),NULL);
    if(not T) return childs;
    vector<vector<QuadtreeNode*> > matrix(int(exp2(K)));
    queue<QuadtreeNode*> q;
    q.push(T);
    //busqueda en anchura donde separamos todos los nodos de T
    //respecto a x en la matriz creada
    while(not q.empty()){
        QuadtreeNode* act= q.front(); q.pop();
        auto w= subtree_to_insert(act->x,x);
        matrix[w].push_back(new QuadtreeNode(act->x));
        for(int j = 0; j<int(exp2(K)); ++j){
            if(act->child[j]){
                q.push(act->child[j]);
            }
        }
    }
    //llamadas a join para cada fila de la matriz que nos devolvera
    //el subarbol resultante para esa fila
    for(int j = 0; j<int(exp2(K)); ++j)
        childs[j] = join(matrix[j]);
    return childs;
}
```

Programa 3 Implementación en `C++` del algoritmo `join`

```
//INPUT: un vector de llaves
//OUTPUT: un arbol randomizado construido a partir de las llaves
QuadtreeNode* join(vector<QuadtreeNode*>& child){
    double max= -1;
    int imax= -1;
    //Busca la llave con prioridad maxima del vector de entrada
    for(int i = 0; i < child.size(); ++i){
        if(child[i] and (child[i]->x)[K] > max ){
            max= child[i]->x[K] ; imax = i;
        }
    }
    if( imax == -1 ) return NULL;
    else{
        //la llave con prioridad maxima se convierte en la raiz
        QuadtreeNode* T = child[imax];
        vector<vector<QuadtreeNode*> > matrix(int(exp2(K)));
        //separamos las llaves respecto a la nueva raiz
        //como haciamos en el algoritmo split
        for(int i = 0; i<int(child.size()); ++i)
            if(i!=imax){
              int w= subtree_to_insert(child[i]->x,child[imax]->x);
              matrix[w].push_back(child[i]);
            }
        //llamamos recursivamente a join para obtener los subarboles
        //de la nueva raiz
        for(int j = 0; j<int(exp2(K)); ++j){
            T->child[j] = join(matrix[j]);
        }
        return T;
    }
}
```

5 Experimentación y análisis experimental

Ahora analizaremos experimentalmente los algoritmos anteriores siempre considerando árboles de n nodos. Posteriormente, compararemos nuestra implementación con los algoritmos de `split` y `join` con una implementación con cola de prioridad y otra sin randomizar. Los experimentos se han realizado en un portátil ASUS con 8GB de memoria RAM, un procesador Intel Core i7-6500U, y Windows 10. Los códigos han sido compilados en gcc versión 6.2.0.

5.1 Costes de los algoritmos

Para cada dimensión k=2,3,4,5,6 calculamos el coste de crear un quad tree k-dimensional T de n nodos y de insertar y borrar llaves con nodos aleatorios e independientes en este. Para ello el programa genera $k + 1$ atributos en el intervalo [0,1], los k primeros corresponden a la llave y el último corresponde a la prioridad. Para cada tamaño n de 25000 a 50000 en incrementos de 5000 se generan 300 árboles de ese tamaño. El programa calcula el número de nodos visitados y el tiempo de ejecución promedio en la creación del árbol. Posteriormente, cuando el árbol es de tamaño n, se insertan y borran llaves del árbol alternativamente para conservar el tamaño del árbol. Se realizan 6000 inserciones y 6000 borrados alternados, es decir, un borrado tras cada inserción y una inserción tras cada borrado. Este proceso se realiza para cada árbol y se cuentan los nodos visitados en la inserción y en el borrado y sus tiempos de ejecución respectivos (independientemente uno del otro). Posteriormente se hace la media de cada inserción y borrado.

Por otra parte, se ha realizado otro experimento para aislar el coste de los algoritmos de `split` y `join`. En ellos para las mismas dimensiones y tamaños del experimento anterior, se generan 50 árboles y se insertan y borran 10 llaves alternadamente. Estas llaves siempre son la raíz del árbol, es decir, cada inserción es de una llave con prioridad más alta que la raíz del árbol y posteriormente se borra esta llave. Se miden los costes en nodos visitados y el tiempo de ejecución como en el otro experimento.

A continuación, vamos a ver los resultados de los experimentos empezando

por el coste de creación, seguido del coste de inserción y en última instancia el coste de borrado. Posteriormente veremos el resultado de los experimentos que aíslan los algoritmos de `split` y `join` y finalmente analizaremos conjuntamente los costes de los algoritmos. Para ello, utilizaremos la notación asintótica clásica [1].

5.1.1 Coste de creación

Aquí tenemos el coste de creación promedio en nodos visitados en la Figura 9 y el tiempo de creación promedio en la Figura 10.

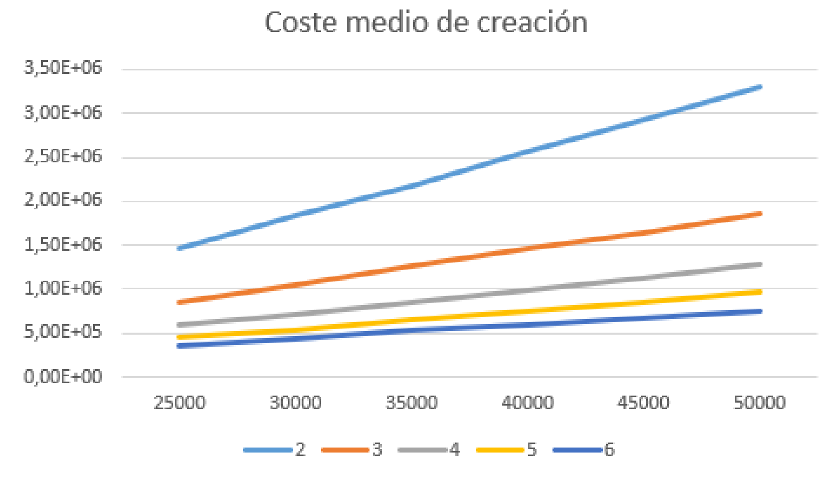


Figura 9: Coste medio de creación en nodos visitados para quad trees de dimensiones k=2,3,4,5,6 y tamaño n de 25000 a 50000

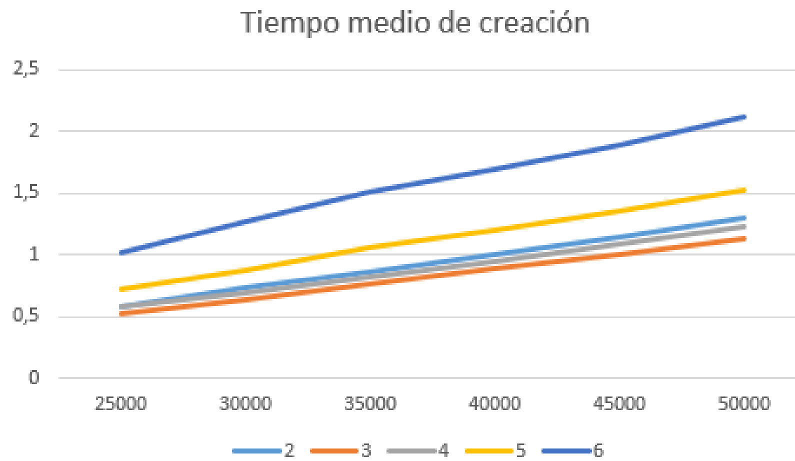


Figura 10: Tiempo medio de creación en segundos para quad trees de dimensiones k=2,3,4,5,6 y tamaño n de 25000 a 50000

Primeramente, en las gráficas podemos apreciar que el coste en nodos visitados es superior cuanto más pequeña es la dimensión. Estos resultados son contrarios a lo esperado intuitivamente. Una posible explicación para este fenómeno es que las constantes de los logaritmos son distintas para dimensiones mayores y esto produce este efecto.

Por otra parte, el tiempo de ejecución parece ser menor para dimensiones 2, 3 y 4. Sorprendentemente la dimensión 3 es la más rápida para este algoritmo. Esto también puede explicarse con las explicaciones anteriores sobre el coste de búsqueda del máximo. Esto probaría que el algoritmo dista de ser óptimo y que posibles optimizaciones pueden hacer que el algoritmo sea más rápido para la dimensión 2 como debería ser intuitivamente.

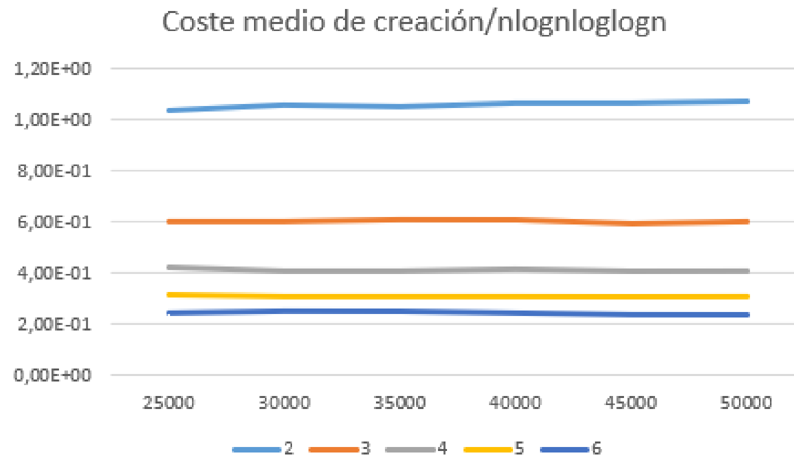


Figura 11: Coste medio de creación en nodos visitados para quad trees de dimensiones k=2,3,4,5,6 y tamaño n de 25000 a 50000 dividido por $n \log(n) \log(\log(n))$

Al dividir el coste por $n \log(n) \log(\log(n))$ nos queda esta gráfica. Dividimos por este valor porque como veremos en la sección de análisis de los costes, además de darnos buenos resultados, este valor concuerda con los demás experimentos. Por tanto, este podría ser el coste de creación de nuestro algoritmo. Más adelante, una vez vistos los demás experimentos, analizaremos en profundidad los costes en conjunto.

5.1.2 Coste de insertado

Aquí tenemos el coste de inserción promedio en nodos visitados en la Figura 9 y el tiempo de inserción promedio en la Figura 10.

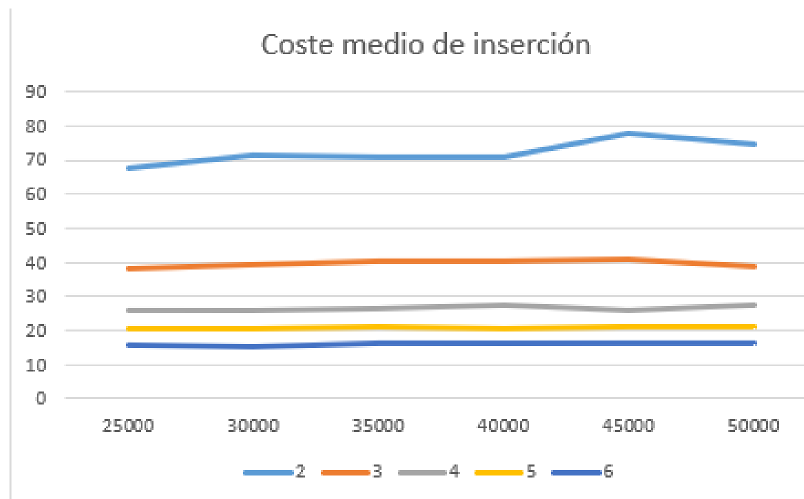


Figura 12: Coste medio de inserción en nodos visitados para quad trees de dimensiones k=2,3,4,5,6 y tamaño n de 25000 a 50000

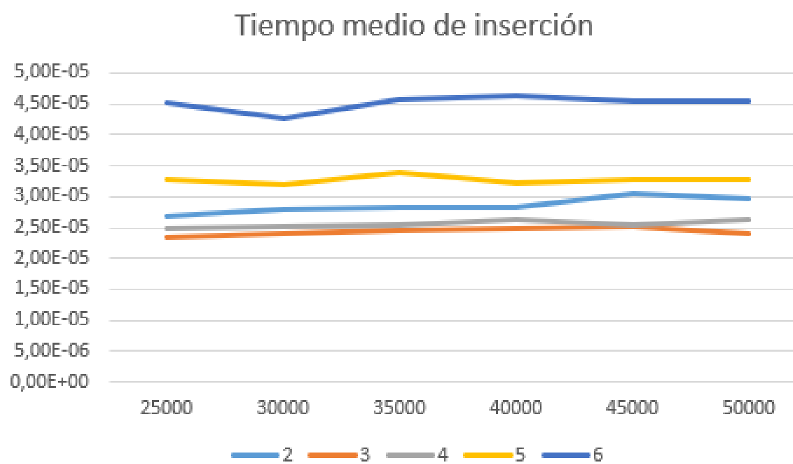


Figura 13: Tiempo medio de inserción en nodos visitados para quad trees de dimensiones k=2,3,4,5,6 y tamaño n de 25000 a 50000

Como podemos ver el coste sigue siendo mayor para dimensión más pequeñas y el tiempo sigue siendo más rápido en dimensión 3.

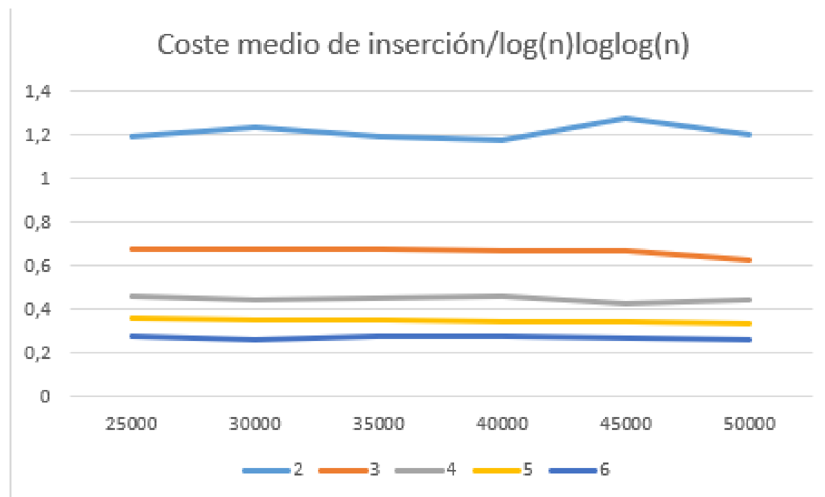


Figura 14: Coste medio de inserción en nodos visitados para quad trees de dimensiones k=2,3,4,5,6 y tamaño n de 25000 a 50000 dividido por $\log(n)\log(\log(n))$

El coste de inserción parece ser del orden $\Theta(\log(n)\log(\log(n)))$ tras dividir los resultados. Esto coincidiría con los resultados del coste de creación ya que la creación consiste en una secuencia de n inserciones.

5.1.3 Coste de borrado

Aquí tenemos el coste de borrado promedio en nodos visitados en la Figura 15 y el tiempo de creación promedio en la Figura 16.

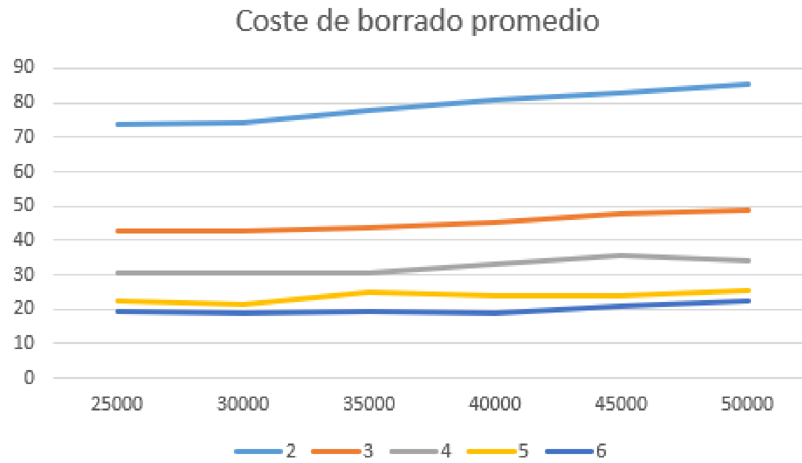


Figura 15: Coste medio de borrado en nodos visitados para quad trees de dimensiones k=2,3,4,5,6 y tamaño n de 25000 a 50000

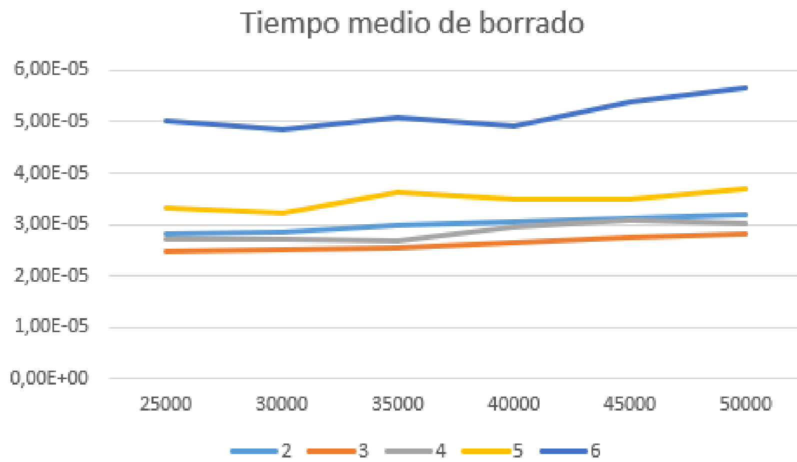


Figura 16: Tiempo medio de borrado en nodos visitados para quad trees de dimensiones k=2,3,4,5,6 y tamaño n de 25000 a 50000

Como podemos ver el coste sigue siendo mayor para dimensiones más pequeñas y el tiempo sigue siendo más rápido en dimensión 3.

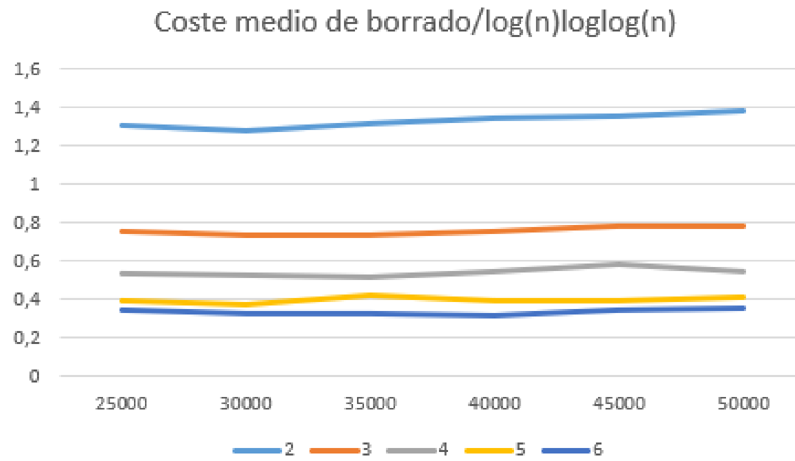


Figura 17: Coste medio de borrado en nodos visitados para quad trees de dimensiones k=2,3,4,5,6 y tamaño n de 25000 a 50000 dividido por $\log(n)\log(\log(n))$

El coste de borrado parece ser del orden $\Theta(\log(n)\log(\log(n)))$ tras dividir los resultados. Esto coincidiría con los experimentos anteriores ya que el coste de borrado es igual que el coste de inserción al usar los mismos algoritmos.

5.1.4 Coste de la inserción en la raíz

Ahora veremos el coste de inserción promedio en nodos visitados cuando el nodo insertado tiene más prioridad que la raíz del árbol en la Figura 18 y el tiempo de inserción promedio en la Figura 19.

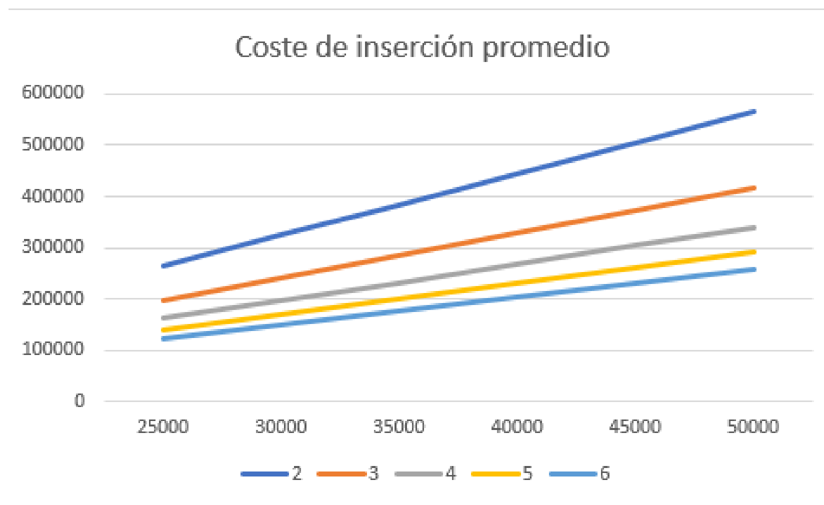


Figura 18: Coste medio de inserción en la raíz en nodos visitados para quad trees de dimensiones k=2,3,4,5,6 y tamaño n de 25000 a 50000

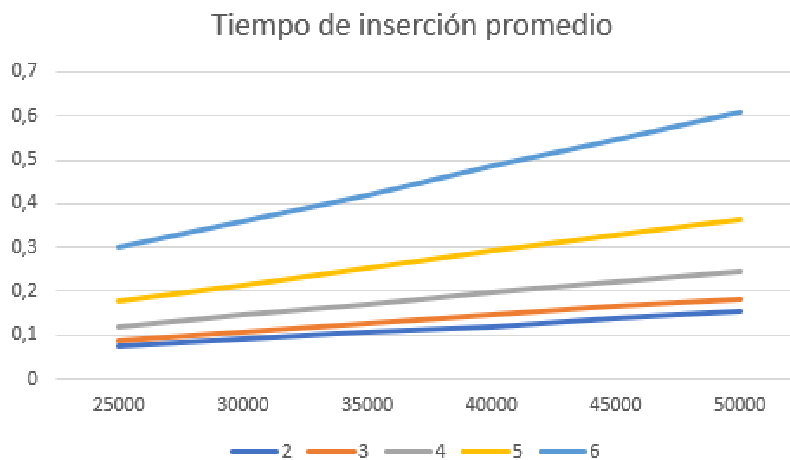


Figura 19: Tiempo medio de inserción en la raíz en nodos visitados para quad trees de dimensiones k=2,3,4,5,6 y tamaño n de 25000 a 50000

En este experimento podemos comprobar que ahora sí el algoritmo más rápido es el de dos dimensiones pese a visitar más nodos.

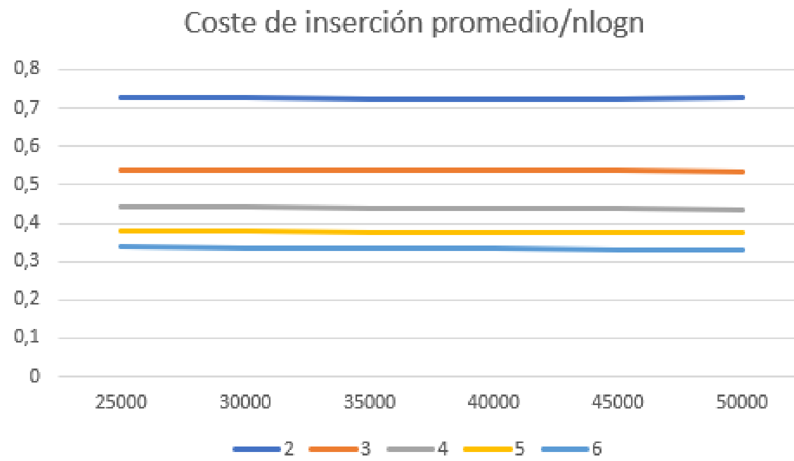


Figura 20: Coste medio de inserción en la raíz en nodos visitados para quad trees de dimensiones k=2,3,4,5,6 y tamaño n de 25000 a 50000 dividido por $n\log(n)$

El coste de inserción parece ser del orden $\Theta(n\log(n))$ tras dividir los resultados. Esto confirmaría los costes conseguidos en los experimentos anteriores, explicaremos cómo posteriormente.

5.1.5 Coste del borrado en la raíz

Ahora veremos el coste de borrado promedio en nodos visitados cuando el nodo insertado tiene más prioridad que la raíz del árbol en la Figura 21 y el tiempo de inserción promedio en la Figura 22.

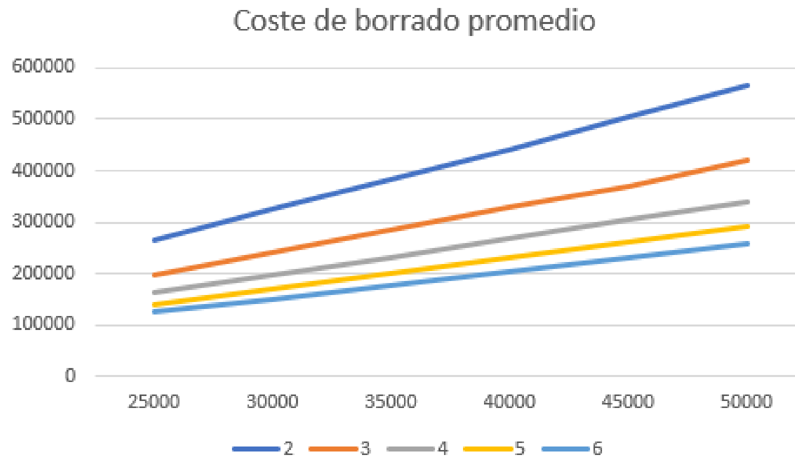


Figura 21: Coste medio de borrado en la raíz en nodos visitados para quad trees de dimensiones k=2,3,4,5,6 y tamaño n de 25000 a 50000

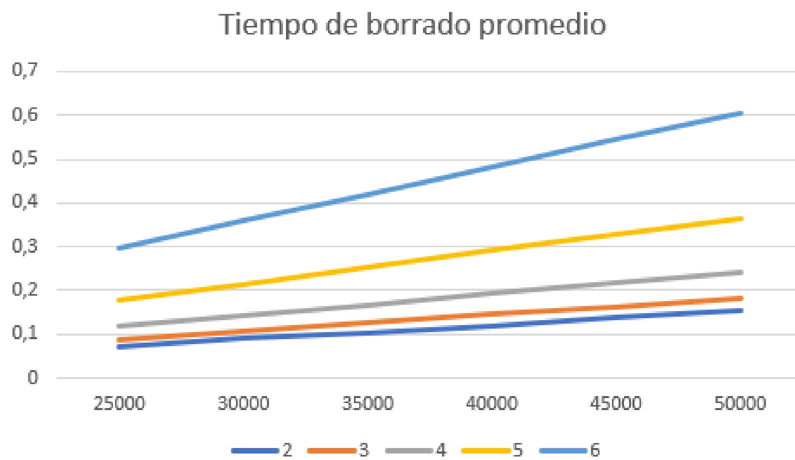


Figura 22: Tiempo medio de borrado en la raíz en nodos visitados para quad trees de dimensiones k=2,3,4,5,6 y tamaño n de 25000 a 50000

Podemos ver que el coste de borrado es muy similar al de inserción.

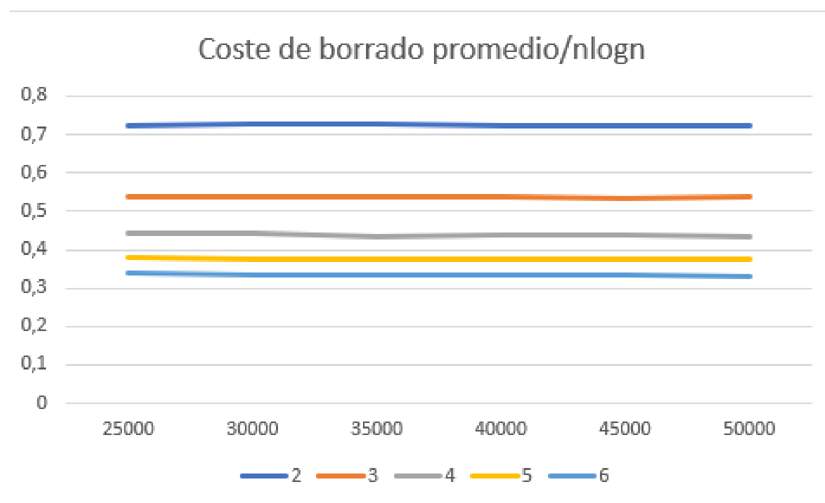


Figura 23: Coste medio de borrado en la raíz en nodos visitados para quad trees de dimensiones k=2,3,4,5,6 y tamaño n de 25000 a 50000 dividido por $n\log(n)$

El coste de borrado parece ser del mismo orden que la inserción, $\Theta(n\log(n))$.

5.1.6 Análisis de los costes

Como hemos visto en los experimentos, los costes de inserción y borrado promedios serian del orden $\Theta(\log(n)\log(\log(n)))$, la creación $\Theta(n\log(n)\log(\log(n)))$ y la inserción y el borrado en la raíz $\Theta(n\log(n))$.

El coste de creación concuerda con el de inserción dado que la creación consiste en una sucesión de n inserciones. Este a su vez concuerda con el coste de borrado ya que utiliza los mismos algoritmos que la inserción.

Por último, el coste de inserción y borrado en la raíz también concuerda con el coste de inserción y borrado promedio ya que como el tamaño promedio de un subárbol elegido al azar de un árbol de tamaño n es del orden de $\Theta(\log(n))$ [9], al substituir n por $\log(n)$ en el coste de inserción en la raíz, $n\log(n)$, nos queda $\Theta(\log(n)\log(\log(n)))$ que es el coste promedio de inserción y borrado promedio que hemos obtenido.

Cabe destacar que intuitivamente creemos que la función que describe el coste promedio (medido como el número de nodos visitados) de los algoritmos randomizados de inserción y de borrado es del orden $\Theta(\log(n) + \log(n)\log(\log(n)))$ para un quad tree de tamaño n. Esto es debido a que ambos algoritmos constan de dos etapas:

- La primera etapa consiste en localizar el sitio del árbol en donde se va a realizar la inserción o el borrado (esto corresponde a recorrer un camino del árbol con coste promedio $\Theta(\log(n))$ donde n es el tamaño del árbol).
- La segunda etapa consiste en aplicar en ese punto el algoritmo de `split` o `join` en subárboles de tamaño promedio $\Theta(\log(n))$ [9].

La combinación de los costes de ambas etapas explica el coste propuesto que además parece verificarse experimentalmente.

5.2 Comparativas

Ahora compararemos nuestra implementación con la implementación con cola de prioridad y la no randomizada.

En la implementación con cola de prioridad los nodos siguen teniendo prioridades, pero a la hora de insertar un nodo en la raíz, los subárboles son pasados a una cola de prioridad e insertados en orden de prioridad nuevamente utilizando el algoritmo de inserción.

En la implementación no randomizada clásica, las inserciones siempre van en las hojas y al borrar se reconstruye el subárbol borrado volviéndolo a insertar en el lugar del nodo eliminado.

En estos experimentos volvemos a utilizar el diseño de experimentos anterior pero ahora también los ejecutamos en las demás implementaciones. Esto significa que para cada dimensión k=2,3,4,5,6 calculamos el coste de crear el árbol, insertar llaves, y borrar llaves. Esto lo hacemos para cada tamaño n de 25000 a 50000 en incrementos de 5000 generando 300 árboles de ese tamaño.

Posteriormente, cuando el árbol es de tamaño n, se realizan 6000 inserciones y 6000 borrados alternados, es decir, un borrado tras cada inserción y una inserción tras cada borrado. Este proceso se realiza para cada árbol y se cuentan los nodos visitados en la inserción y en el borrado y sus tiempos de ejecución respectivos. Posteriormente se hace la media de cada inserción y borrado.

5.2.1 Coste de creación

Ahora veremos el coste en nodos visitados y en segundos de ejecución durante la creación de los árboles para las dimensiones de la 2 a la 6 separadamente.

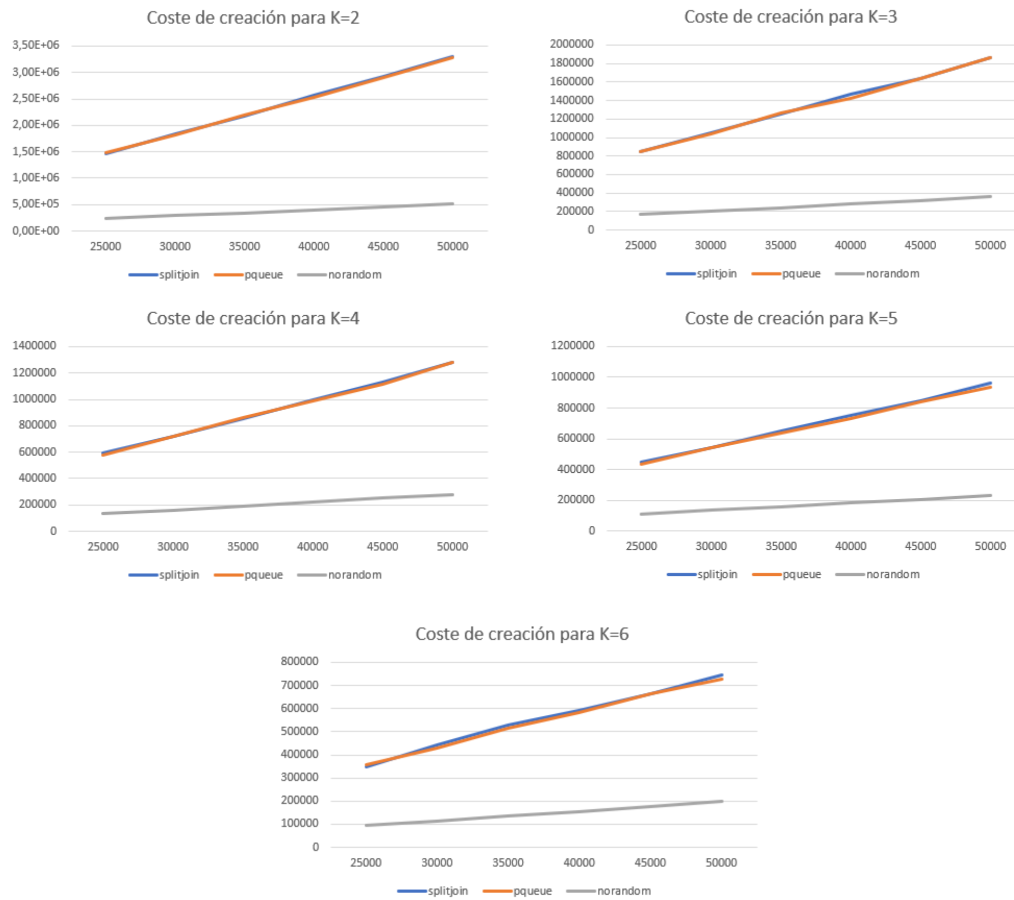


Figura 24: Coste medio de creación en nodos visitados para quad trees de dimensión k=2,3,4,5,6 y tamaño n de 25000 a 50000 para los tres algoritmos distintos

Como podemos ver el coste en nodos visitados entre nuestro algoritmo y el de la cola de prioridad parece ser el mismo mientras que el coste de la implementación no randomizada es considerablemente menor.

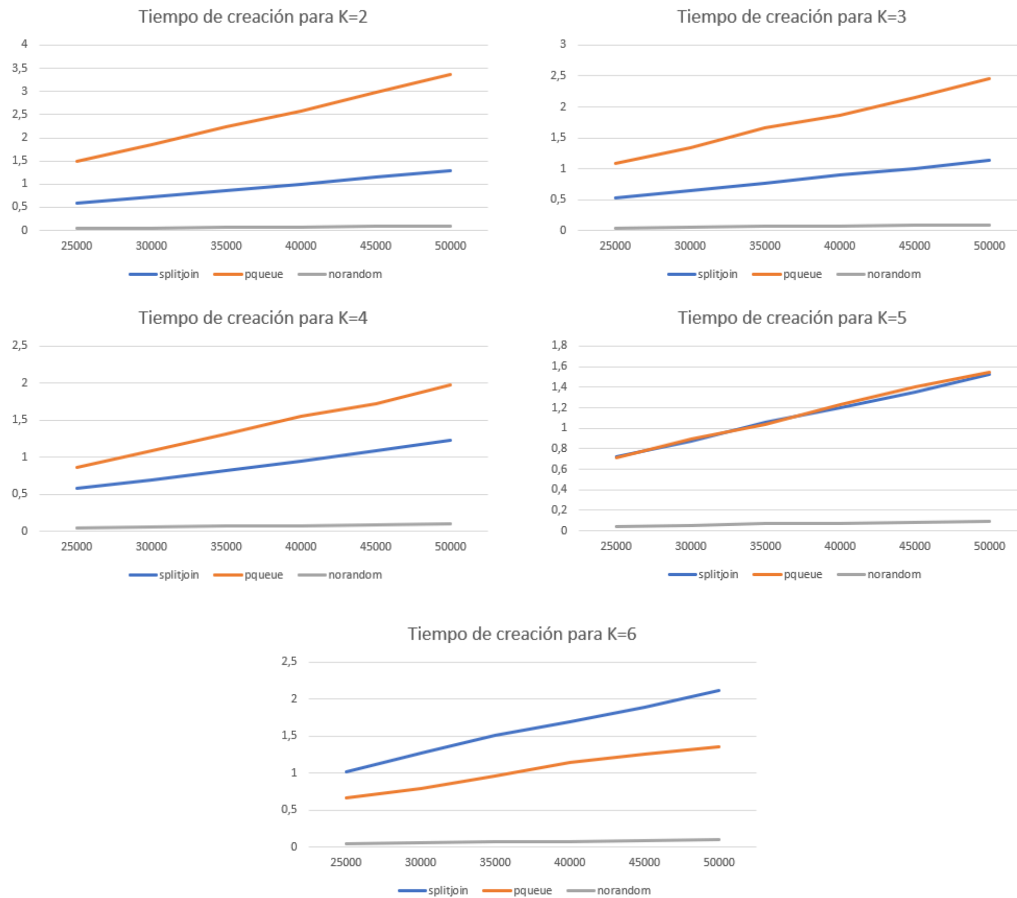


Figura 25: Tiempo medio de creación en segundos para quad trees de dimensión k=2,3,4,5,6 y tamaño n de 25000 a 50000 para los tres algoritmos distintos

El tiempo de ejecución sigue siendo inferior en la no randomizada pero podemos observar como nuestra implementación respecto a la de la cola de prioridad es más veloz para las dimensiones de la 2 a la 4 y más lenta en la dimensión 6.

Esto puede deberse a que los nodos que deben ser remplazados tienen que atravesar caminos más largos cuando la dimensión es más pequeña en el algoritmo de cola de prioridad mientras que no es el caso en la nuestra.

5.2.2 Coste de inserción

Ahora veremos el coste en nodos visitados y en tiempo de CPU en segundos durante la inserción de los árboles para las dimensiones de la 2 a la 6 separadamente.

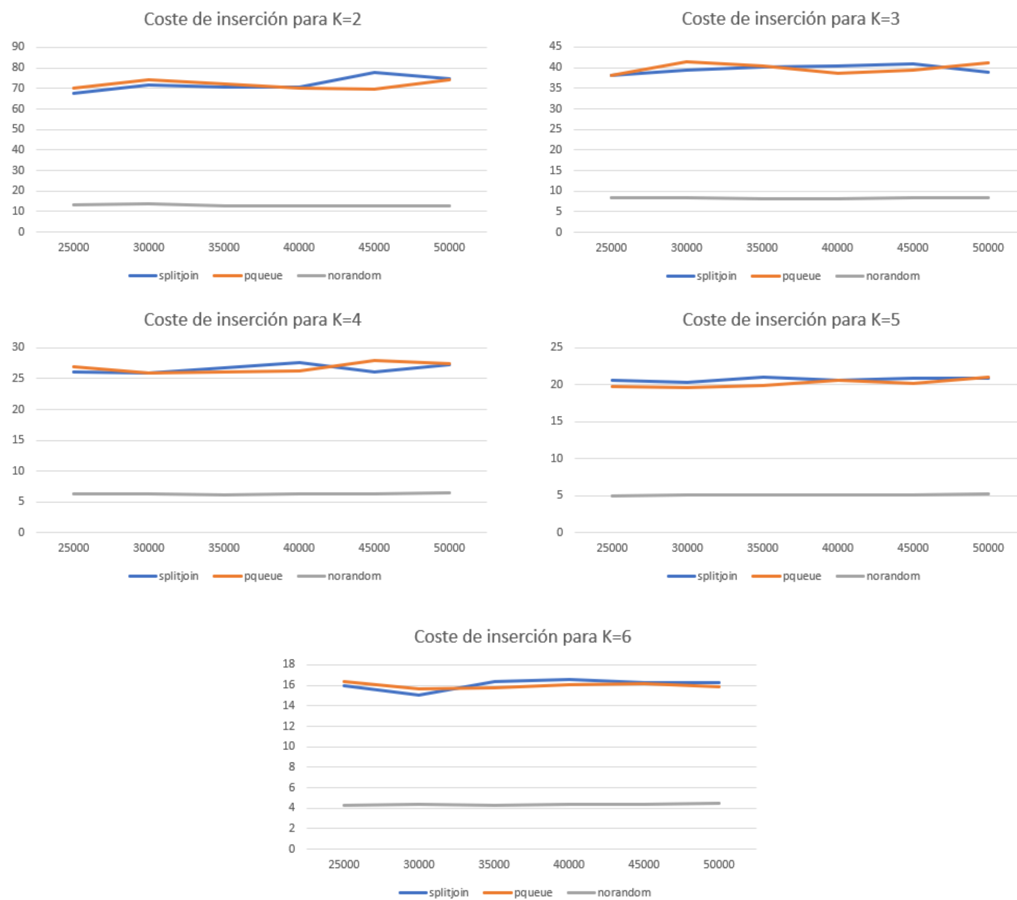


Figura 26: Coste medio de inserción en nodos visitados para quad trees de dimensión k=2,3,4,5,6 y tamaño n de 25000 a 50000 para los tres algoritmos distintos

Otra vez, el coste entre nuestros algoritmos y la cola de prioridad parece ser idéntico.

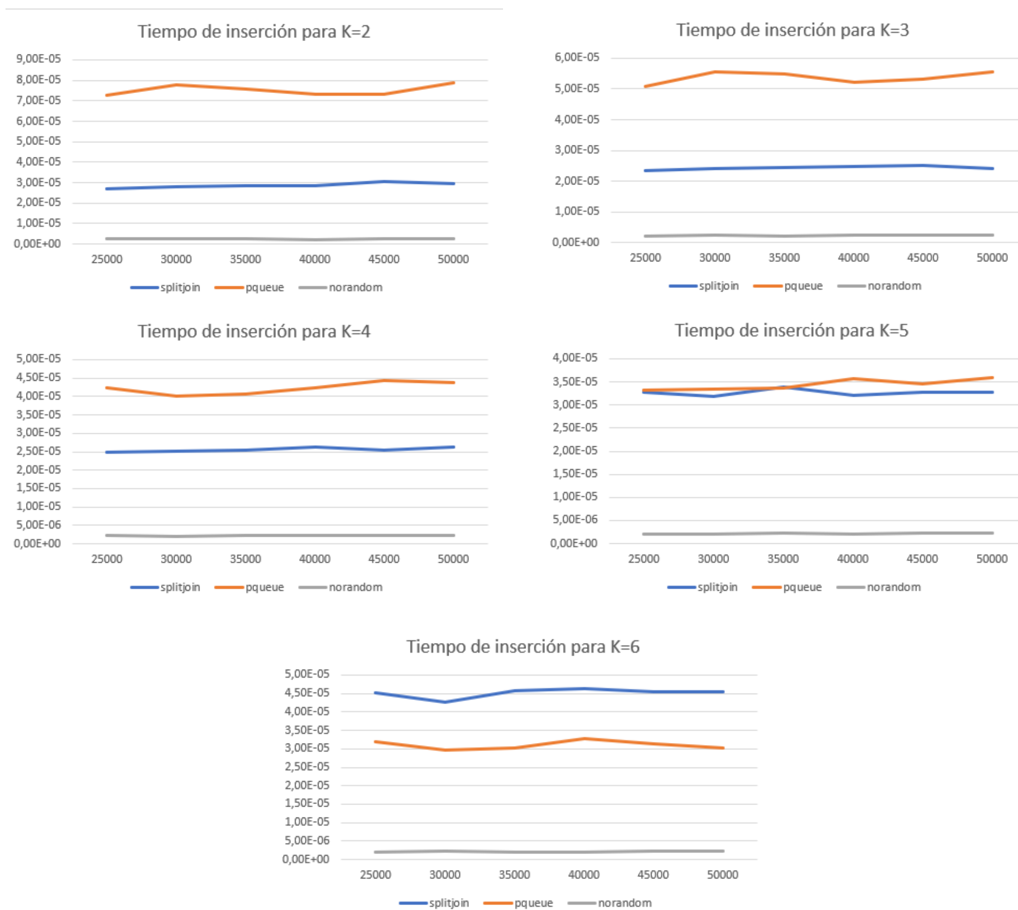


Figura 27: Tiempo medio de inserción en segundos para quad trees de dimensión k=2,3,4,5,6 y tamaño n de 25000 a 50000 para los tres algoritmos distintos

Similarmente, el tiempo de inserción se comporta como el tiempo de creación. No es nada extraño ya que la creación consiste en una sucesión de n inserciones para distintos valores de n.

5.2.3 Coste de borrado

Por último, veremos el coste en nodos visitados y en segundos de CPU durante el borrado de los árboles para las dimensiones de la 2 a la 6 separadamente.

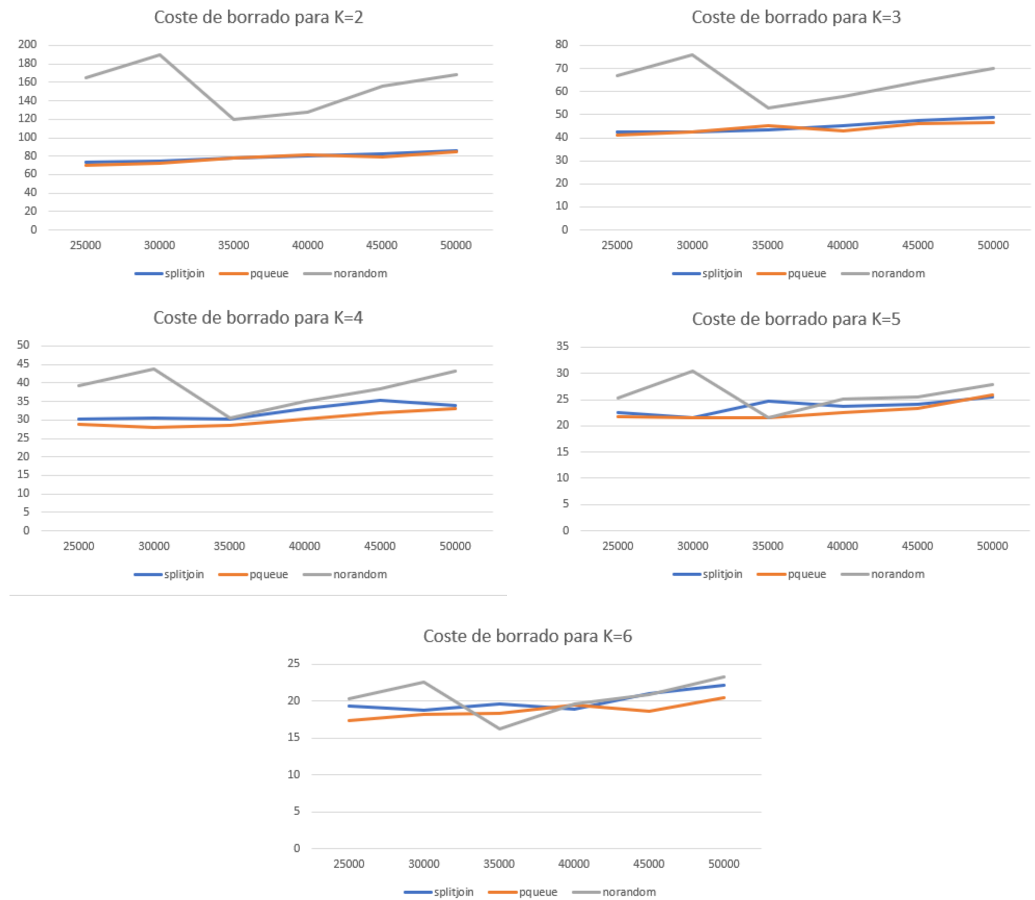


Figura 28: Coste medio de borrado en nodos visitados para quad trees de dimensión k=2,3,4,5,6 y tamaño n de 25000 a 50000 para los tres algoritmos distintos

Aquí podemos ver que los algoritmos randomizados se comportan mejor que el no randomizado, sobre todo para dimensiones pequeñas. El coste

de borrado parece ser similar para los algoritmos randomizados, ligeramente beneficiando al de cola de prioridad al aumentar la dimensión.

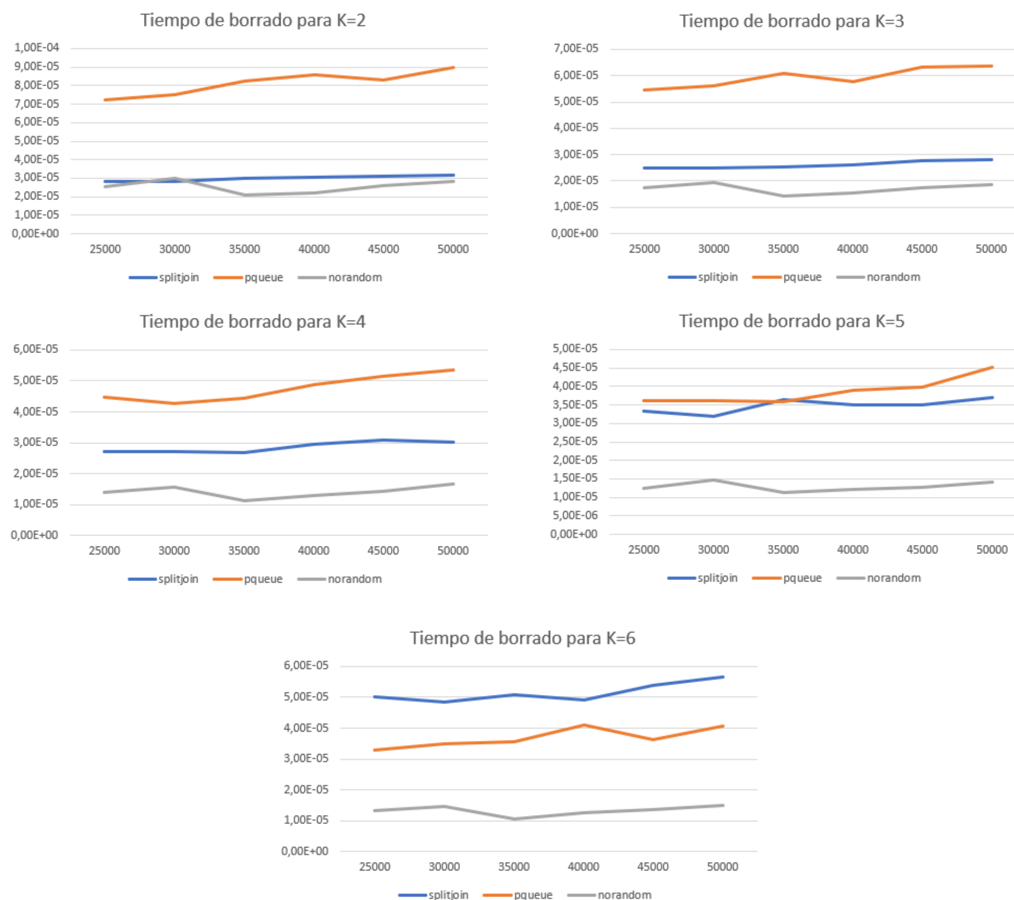


Figura 29: Tiempo medio de borrado en segundos para quad trees de dimensión k=2,3,4,5,6 y tamaño n de 25000 a 50000 para los tres algoritmos distintos

Pese a tener un coste mayor en nodos visitados, el algoritmo de borrado clásico parece ser el más veloz. El nuestro llega a ser comparable a este para dimensiones pequeñas pero su ventaja se va perdiendo al aumentar la dimensión hasta ser más lento que el de cola de prioridad como hemos visto en las demás comparaciones.

Cabe destacar que los experimentos se han realizado con nodos aleatorios como input. Si el input fuese ordenado el coste del algoritmo no randomizado sería muy superior a lo que hemos visto mientras que los randomizados no serían afectados de esa manera.

6 Planificación y análisis económico

6.1 Planificación

En este capítulo mostraremos la planificación inicial y final y describiremos los cambios ocurridos. A continuación, en la Figura 30 tenemos la planificación inicial y en la Figura 31 tenemos la final.

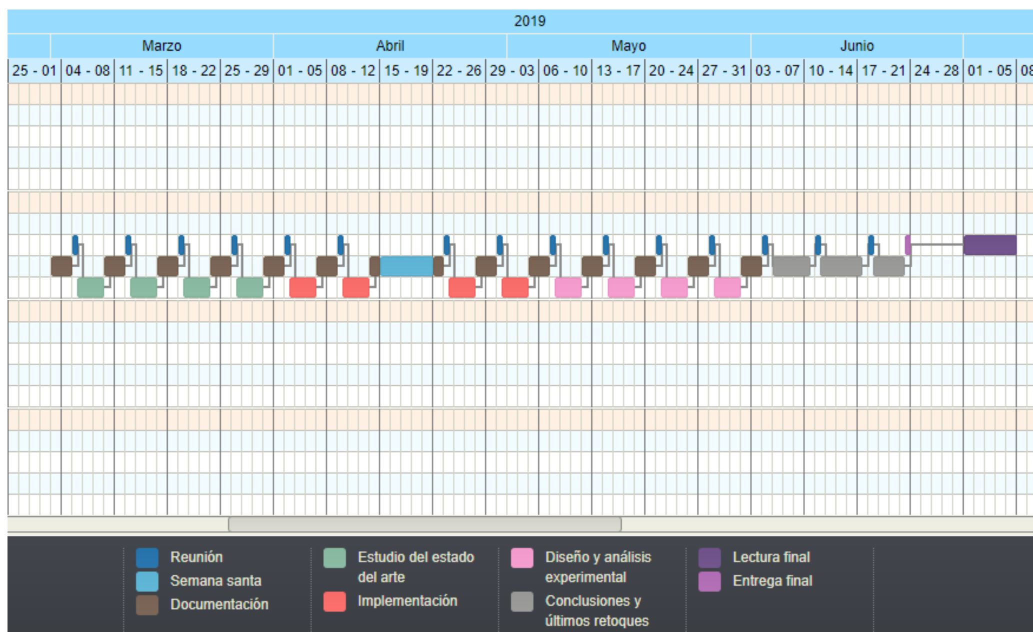


Figura 30: Planificación inicial por semanas y tareas.

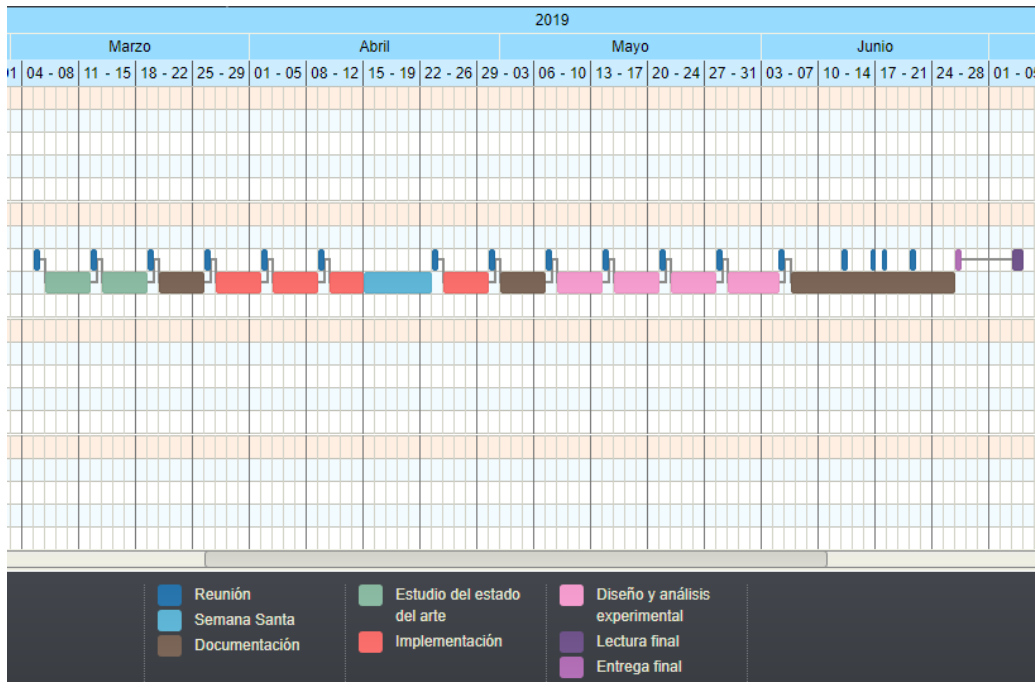


Figura 31: Planificación final por semanas y tareas.

El primer cambio que podemos apreciar es el movimiento de la tarea de documentación al final de cada etapa. Nos dimos cuenta de que el trabajo que se podía avanzar documentando no era tan grande y que era necesario tener otras partes acabadas.

Por otra parte, la implementación y el diseño y análisis de experimentos ha supuesto más tiempo del esperado. La implementación supuso primeramente implementar la propuesta original y al ver que no era eficiente se invirtieron un par de semanas en optimizar los códigos. Por otra parte hubo que implementar las otras dos propuestas y preparar todas las implementaciones para la fase experimental. El análisis de experimentos tuvo que ser repetido dado que la muestra inicialmente elegida no aportaba datos precisos. Por otra parte también se hicieron varias pruebas e experimentos que no dieron resultados concluyentes.

Pese a estos retrasos, la fase de Estudio del estado del arte fue más rápida de los esperado y las horas de conclusiones y últimos retoques (cuyo objetivo

era representar un tiempo para cubrir posibles imprevistos) fueron suficientes para ser reasignadas a estas fases haciendo que el proyecto se completase en el periodo de tiempo esperado de 465h.

Tarea	Horas estimadas	Horas reales
Estudio del estado del arte	72	60
Implementación	72	120
Diseño y análisis experimental	72	120
Conclusiones y últimos retoques	78	0
Documentación	156	150
Reuniones	15	15
Total	465	465

6.2 Análisis económico

Ahora calcularemos el coste final del proyecto y las desviaciones respecto al presupuesto inicial.

El presupuesto inicial del proyecto era de 21620€. De este presupuesto lo único que ha sido modificado han sido la distribución de las tareas realizadas.

La distribución de las horas finalmente queda de la siguiente manera:

Fase	J. de Proyecto	Desarrollador	Analista	Total
Estudio del estado del arte	60h	0h	0h	60h
Implementación	20h	100h	0h	120h
Diseño y análisis experimental	20h	0h	100h	120h
Documentación	150h	0h	0h	150h
Reuniones	15h	0h	0h	15h
Total	265h	100h	100h	465h

Después de asignar el coste de hora correspondiente a cada tarea nos queda el coste total de personal:

Rol	Coste/hora	Horas	Coste
Jefe de proyecto	50€/h	265h	13250€
Desarrollador	35€/h	100h	3500€
Analista	30€/h	100h	3000€
Total			19750€

Los demás costes se mantienen igual. El coste de Hardware consiste en un portátil y el Software es de libre distribución. Los costes indirectos incluyen un espacio que incluye los costes de Internet, electricidad y mobiliario. Añadiendo estos costes nos queda el siguiente coste total:

Concepto	Coste
Hardware	50€
Software	0€
Recursos humanos	19750€
Costes indirectos	320€
Total	20120€

Ahora sí, podemos ver que la desviación del coste del proyecto ha sido de 1500€ por debajo del presupuesto y la desviación en consumo ha sido nula dado que se han invertido el mismo número de horas que el esperado.

7 Sostenibilidad

7.1 Dimensión ambiental

El proyecto ha tenido un impacto ambiental mínimo. Se ha usado hardware reacondicionado para minimizar el daño ambiental que producen los deshechos tecnológicos y la producción de nuevos dispositivos. Por otra parte, el consumo de electricidad ha sido tenido en cuenta.

Si el ordenador consume una media de 100W y necesitamos 465h, entonces eso son 46,5KWh que equivalen a 17,9kg de CO2. Es una gran cantidad de energía pero es la necesaria para el desarrollo del proyecto.

Durante la vida útil del proyecto no se usarán más recursos por nuestra parte. El proyecto ha tenido como finalidad publicar los algoritmos para el uso libre de estos y así reducir los costes que conllevarían a empresas e individuos reproducir los resultados. Todo ello conlleva una reducción del impacto negativo al medio ambiente.

7.2 Dimensión económica

Ya se ha presentado un análisis completo del presupuesto del proyecto. En la propuesta se ha tenido en cuenta los costes materiales al elegir software libre y hardware reacondicionado. El coste principal son recursos humanos.

El problema actual se resuelve gastando recursos de manera privada en cada situación que se necesita. Mi propuesta ofrece una alternativa gratuita. Esto comporta un ahorro significante a los usuarios que no tienen que invertir más recursos humanos de los necesarios. Por tanto, el ahorro futuro compensa esta inversión actual. Durante la vida útil del proyecto no habrán más costes económicos por nuestra parte.

7.3 Dimensión social

Este proyecto me ha aportado la capacidad de emprender proyectos y llevarlos de una forma correcta. Me ha enseñado a planificar el tiempo y estimar los recursos. Así como ver que imprevistos surgen durante la realización.

Como ya hemos dicho el problema actual se resuelve de forma individual y privada. Nuestra propuesta permite a los usuarios ganar tiempo al tener los algoritmos públicos. Por otra parte, el hecho de realizar el proyecto beneficia directamente a la comunidad de investigación. Además, el proyecto no perjudica a ningún usuario ni les produce ninguna dependencia.

8 Conclusiones y trabajo futuro

8.1 Conclusiones

En este proyecto hemos intentado implementar la propuesta *Randomized insertion and deletion in point quad trees* [4] y analizar experimentalmente su coste, especialmente el del borrado de nodos, y compararlo con el de algoritmos clásicos.

Lo primero que vimos es que la implementación tal como estaba descrita era muy ineficiente en la práctica y no podía competir con los algoritmos clásicos.

Basándonos de los algoritmos propuestos en dicho artículo se realizó una implementación alternativa que puede ser optimizada en un futuro.

Al realizar los experimentos vimos que los costes de inserción y borrado de esta implementación podrían ser del orden $\Theta(\log(n)\log(\log(n)))$ mientras que el coste de los algoritmos aplicados a la raíz del árbol podrían ser del orden $\Theta(n\log(n))$.

Al comparar la implementación con una implementación basada en una cola de prioridad vemos que el coste asintótico parece ser el mismo y que temporalmente podría ser más rápida nuestra implementación para dimensiones menores a 5.

Al comparar la implementación con una implementación clásica sin randomizar vemos que el coste de nuestra implementación es menor en el borrado, pero cuando se mide el tiempo de ejecución, la implementación clásica tarda un tiempo similar.

Aun así, en nuestros experimentos los nodos insertados y borrados son aleatorios. Si no fuese así, el coste del algoritmo clásico seria mayor y nuestra implementación podría ser más rápida, sobre todo si el input estuviese ordenado.

8.2 Trabajo futuro

A partir de estas conclusiones, vemos que queda mucho trabajo interesante para hacer en este campo.

Primeramente, nuestra implementación propuesta en este proyecto podría ser mejorada. Por ejemplo, el algoritmo `join` elige el máximo linealmente y quizás se podría aprovechar alguna estrategia diferente para ahorrar costes. Además, se podrían aprovechar algunas propiedades de la definición de los quad trees (que aún no han sido aprovechadas en nuestra implementación) para ahorrar algunas comparaciones entre llaves.

Aunque no era el objetivo inicial de este trabajo, podrían estudiarse experimentalmente las varianzas de los costes de inserción y borrado y hacer un análisis estadístico en mayor profundidad.

También sería muy interesante obtener el coste exacto de nuestras operaciones de inserción y borrado obtenido mediante cálculos matemáticos formales. Sin embargo, puede ser un problema difícil dado que las herramientas matemáticas (combinatoria analítica) que se requieren para este tipo de análisis no son básicas.

Dado que nuestros experimentos han trabajado con nodos insertados en orden aleatorio, podría probarse cuál es el coste de tener un input ordenado y comparar los algoritmos randomizados aquí propuestos contra la implementación no randomizada usando un preprocesamiento del input que alterase el orden del input y ver si el tiempo de ejecución sigue siendo mejor.

Podría ser también interesante ver cómo se desempeñan nuestros algoritmos con datos de aplicaciones reales, por ejemplo, datos provenientes de bases de datos genómicas que además de contener grandes cantidades de puntos, son altamente multidimensionales (dimensión 19 en algunos casos) [6].

Para tener acceso al código y probar experimentos o mejoras, aquí se encuentra el repositorio:

`https://bitbucket.org/Oliver_Mp0/randomized-quad-trees/src/master/`

Referencias

[1] T. H. Cormen. *Introduction to algorithms*. MIT Press, 2001.

[2] J. C. Culberson. *The effect of updates in binary search trees*. STOC '85 Proceedings of the seventeenth annual ACM symposium on Theory of computing, 205-212, 1985.

[3] L. Devroye and J. M. Robson. *On the Generation of Random Binary Search Trees*. SIAM J. Comput., 24(6), 1141–1156, 1995.

[4] A. Duch. *Randomized insertion and deletion in point quad trees*. Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de, Catalunya, Barcelona, Spain, 2004.

[5] J. L. Eppinger. *An empirical study of insertion and deletion in binary search trees*. Communications of the ACM, 26(9):663-669, 1983.

[6] 1000 Genomes Project Consortium et al. 2015. *A global reference for human genetic variation*. 7571, 68-74, 2015.

[7] R. A. Finkel and J. L. Bentley. *Quad trees: a data structure for retrieval on composite key*. Acta Informatica, 4(1):1–9, 1974.

[8] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching, volume 3*. Addison–Wesley, 2nd edition, 1998.

[9] C. Martínez, A. Panholzer, and H. Prodinger. *On the number of descendants and ascendants in random search trees*. Electronic Journal on Combinatorics, 5(1), 1998.

[10] C. Martínez and S. Roura. *Randomized binary search trees*. Journal of the ACM, 45(2):288-323, 1998.

[11] C. McGeoch. *A Guide to Experimental Algorithmics*. Cambridge University Press, 2012.

[12] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[13] H. Samet. *Deletion in two-dimensional quad-trees*. Communications of the ACM, 23(12):703–710, 1980.

[14] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, 1990.

[15] H. Samet. *Foundations of Multidimensional and Metric Data Structures.* Morgan & Kaufman Publ., 2006.

[16] R. Seidel and C. Aragon. *Randomized search trees.* Algorithmica 16, 464–497, 1996.