Centro de Investigación en Matemáticas, A.C.

Diseño e Implementación de un B+-Tree para Android

REPORTE TÉCNICO

Que para obtener el grado de

Maestro en Ingeniería de Software

P r e s e n t a

I.S.C Oscar Fabricio Valdez Castillo

Director de Reporte Técnico

Dr. Jorge Manjarrez Sánchez

Zacatecas, Zacatecas., 14 de 08 de 2012

Agradecimientos

Quisiera dedicar las siguientes líneas a las personas que hicieron posible la realización del presente documento: en primera instancia a mi madre **Lourdes Valdez Castillo** por otorgarme la vida y preocuparse siempre por mi educación y bienestar general, al M. en C. Isaac Guzmán Domínguez, Director de la Unidad Profesional Interdisciplinaria de Ingeniería Campus Zacatecas (UPIIZ) por haberme brindado la oportunidad de asistir a clases para obtener el grado.

A todo el personal del Centro de Investigaciones en Matemáticas A. C., Unidad Zacatecas (CIMAT) por haberme dedicado tiempo, paciencia y a la vez inculcar una titánica motivación para seguir adelante; en especial al Dr. Jorge Manjarrez Sánchez, director de mí reporte técnico, por transmitirme su conocimiento y ser además, un gran amigo.

Al Consejo Zacatecano de Ciencia y Tecnología (COZCyT) y al Centro de Investigación en Matemáticas A. C., Unidad Zacatecas (CIMAT) como instituciones, por haber financiado la realización de mis estudios a lo largo de la maestría en Ingeniería de Software.

A todos los anteriores GRACIAS, por creer en mí y brindar una mentalidad sana en aras del conocimiento, acrecentando el cariño por lo que me gusta hacer.

Resumen

Una de las tareas fundamentales del cómputo es el procesamiento de datos. A lo largo de la historia de la Ciencia de la Computación se han propuesto diversas estructuras como el B+-Tree, que permiten el almacenamiento y búsqueda de datos de manera eficiente. Actualmente, el desarrollo tecnológico de los dispositivos móviles los hace una plataforma interesante para estas tareas y el desarrollo de aplicaciones tipo data-driven, sin embargo, a pesar de los progresos tecnológicos, siguen existiendo restricciones en cuanto a su capacidad de memoria, poder de procesamiento y consumo de energía. El presente trabajo describe las consideraciones de diseño e implementación de un B+-Tree para el sistema operativo Android, consideraciones necesarias para superar en lo posible las limitaciones de los dispositivos móviles para así obtener un rendimiento óptimo y aprovechar al máximo las características técnicas del dispositivo y la libertad de la movilidad que ofrece.

Como propuesta inicial hacia una solución eficiente, se realizaron tres implementaciones que atienden al problema mencionado utilizando un mecanismo de persistencia basado en la serialización. Una de las implementaciones muestra tener un mejor rendimiento en un conjunto de datos moderado, en comparación con una implementación comercial existente.

Abstract

One of the main tasks of computing is data processing. Through the history of Computer Science, there have been proposed several structures to efficiently store and retrieve data, such as the B+-Tree. Nowadays, the technological development of mobile devices offers an interesting platform for these tasks and the development of data-driven apps, however, in despite of the technological progress; some limitations remain on the memory capacity, the processing power and battery life. This work describes the design and implementation considerations of a B+-Tree on Android, required to overcome these constraints and to achieve optimal performance and, to take advantage of the mobility.

This is an ongoing work towards an efficient B+-Tree implementation for mobile platforms. Three proof of concept implementations were done with a serialization based persistent storage. One of those implementations shows better performance over a moderate data set compared with a commercial implementation.

Índice

Resumen 5
Abstract 5
1. Capítulo Primero: Introducción 13
2. Capítulo Segundo: Los árboles 14
2.1 Conociendo el B-Tree 14
2.2 Estructura de un B-Tree 16
2.1 Algoritmos 20
2.1.1 Búsqueda 20
2.1.2 Búsqueda por rango 22
2.1.3 Inserción 23
2.1.4 Eliminación 24
2.2 Variaciones del B-Tree 28
2.3 El B+-Tree 33
3. Capitulo Tercero: Arquitectura y mejores prácticas en Android 36
3.1 Requisitos mínimos en la plataforma móvil Android 36
3.2 Arquitectura Android 37
3.3 Ejecución del Software y consideraciones sobre la plataforma Android 38
3.3.1 Máquina Virtual Dalvik 38
3.3.2 Tipos de memoria 41
3.3.3 Ejecución de aplicaciones en Android 41
3.4 Eficiencia en las aplicaciones Android 42
3.4.1 Optimización de código 42
3.4.2 Uso de NDK (Native Development Kit) 43
3.4.3 Ciclo de vida de una aplicación Android 44
3.4.4 Utilizando la memoria eficientemente 45
3.4.5 Overhead 46
4. Capítulo Cuarto: Métodos de Persistencia 48
4.1 Análisis de implementación de Persistencia 48
4.2 API de persistencia en Java 48

4.3 Serialización de Objetos 48
4.4 Persistencia a bajo nivel 49
4.5 Contraste de métodos para implementar Persistencia 49
5. Capitulo Quinto: Diseño e implementación de un B+-Tree 52
5.1 Implementaciones basadas en Serialización 52
5.1.1 Diagrama de componentes 52
5.1.2 Diagrama de Clases 54
5.1.3 Clase Nodo 54
5.1.4 Clase Nivel 55
5.1.5 Clase NivelController 56
5.1.6 Clase ArbolMainController 57
5.1.7 Clase ListaDoble 58
5.1.8 Clase Activity 58
5.1.9 Actividades 59
5.1.10 Persistencia 62
5.1.11 Implementación 63
5.1.12 Implementando el algoritmo de búsqueda 63
5.1.13 Implementando el algoritmo de inserción 66
5.1.14 Implementando el algoritmo de eliminación 67
5.1.15 Implementación de la persistencia 68
5.2 Implementación por escritura de archivos 71
5.2.1 Clases 71
5.2.2 Implementación de la persistencia 72
5.2.3 Implementación de búsqueda 75
5.2.4 Implementación de inserción 77
5.2.5 Implementación de la eliminación 78
5.3 Utilización del árbol 79
5.3.1 Inserción 79
5.3.2 Eliminación 79
5.3.3 Búsqueda 79
5.3.4 Búsqueda por valor 79
5.3.5 Búsqueda por rango 79

5.3.6 Diseño de pantallas compartidas 80
6. Capítulo Sexto: Evaluación del Performance 83
6.1 Datos de prueba para implementaciones 83
5.1.1 Generador de datos 83
6.2 Tiempos de ejecución en las operaciones del B+-Tree 84
6.2.1 Velocidad de inserción 85
6.2.2 Velocidad de consulta 87
6.2.3 Velocidad de eliminación 88
6.2.4 Capturas de Pantalla representativas en caso de inserción 89
6.3 Recursos del dispositivo (Memoria) 90
6.4 Tamaño de los archivos 93
7. Capítulo Séptimo: Conclusiones y trabajo futuro 95
Referencias 97
Referencias Web 100

Índice de Figuras

Figura 1. Estructura de un B-Tree 16
Figura 2. Ejemplo de un B-Tree 17
Figura 3. Balanceo en profundidad en un B-Tree 17
Figura 4. Lógica B+-Tree 18
Figura 5 Búsqueda sencilla B+-Tree 21
Figura 6 Búsqueda por rango B+-Tree........ 22
Figura 7. Inserción en un B+-Tree 24
Figura 8. Eliminación sencilla 26
Figura 9. Eliminación con re-distribución........ 26
Figura 10. Eliminación B+-Tree Mezclando niveles........ 27
Figura 11. Copy-up 34
Figura 12. Push up 34
Figura 13. Arquitectura de la plataforma Android [Ehringer 2010] 37
Figura 14. Diferencias entre JME y Dalvik en su estructura de archivos [Eringer 2010]........ 39
Figura 15. Anatomía de un archivo Dex 39
Figura 16. Comparación del tamaño entre archivos compilados java 40
Figura 17. Ciclo de vida de una aplicación Android 44
Figura 18. Diagrama de Componentes B+-Tree Android 52
Figura 19. Diagrama de Clases B+-Tree Android para implementaciones por serialización........ 54
Figura 20. Insertando en un B+-Tree........ 59
Figura 21. Búsqueda sencilla 60
Figura 22. Búsqueda por rango 60
Figura 23. Eliminación de un nodo B+-Tree 61
Figura 24. Implementación con nodo serializado 62
Figura 25. Serialización de todo el B+-Tree........ 62
Figura 26. Alias para encontrar el nivel a insertar (ArbolMainController.java) 63
Figura 27. Función encontrar Nivel (Nivel.java) 64
Figura 28. Alias para encontrar un nodo (Nivel.java)........ 64
Figura 29. Encontrando un nodo en el nivel (Nivel.java) 65
Figura 30. Método alias para búsqueda por rango (ListaDoble.java) 65
Figura 31. Función búsqueda por rango (ListaDoble.java)........ 65
Figura 32. Función alias insertar (ArbolMainController.java) 66
Figura 33. Función insertar (NivelController.java) 66
Figura 34. Inserción cuando existe overflow (NivelController.java) 67
Figura 35. Método alias para eliminar (ArbolMainController.java)........ 67
Figura 36. Parte algoritmo de eliminación (NivelController.java) 68
Figura 37. Parte algoritmo eliminación (NivelController.java) 68

Figura 38. Implementación de la persistencia por nodos serializandolos de manera independiente (Persistencia.java) 69
Figura 39. Implementación de lectura de nodos serializados (Persistencia.java) 69
Figura 40. Implementación de la serialización en todo el árbol 70
Figura 41. Deserialización del B+-Tree (Persistencia.java) 70
Figura 42. Diagrama de clases, implementación de persistencia a bajo nivel 71
Figura 43. Estructura del archivo de la persistencia 71
Figura 44. Clase Persistencia tercer_implementacion/Persistencia.java 72
Figura 45. Lector en tercera_implementacion/Lector.java 72
Figura 46. Singleton para manejar archivo FileSingleton.java 73
Figura 47. Función para reconstruir Nodos a partir de un bufferPagina.java 73
Figura 48. Reconstrucción de un nodo por medio de buffer tercera_implementacion/Nodo.java . 74
Figura 49. Escritor tercer_implementacion/Escritor.java 74
Figura 50. Algoritmo búsqueda tercer_implementacion/FBTree.java 75
Figura 51. Función que busca la siguiente página tercer_implementacion/Pagina.java 75
Figura 52. Buscar Nodo (Pagina.java) 76
Figura 53. Inserción B+-Tree FBTree.java 77
Figura 54. Implementación de la eliminación FBTree.java 78
Figura 55. Diagrama de casos de uso B+-Tree Android 80
Figura 56. Pantalla insertar 80
Figura 57. Pantalla principal 80
Figura 58. Pantalla de inserción 81
Figura 59. Pantalla de eliminación 81
Figura 60. Pantalla selección método de búsqueda 81
Figura 61. Pantalla de búsqueda por índice 82
Figura 62. Búsqueda por valor 82
Figura 63. Búsqueda por rango 82
Figura 64. Función que genera un registro aleatorio (DataGenerator.java) 84
Figura 65. Comparación en operación de inserción 86
Figura 66. Comparación en operación consulta 87
Figura 67. Comparación en operación de eliminación 88
Figura 68. Capturas de pruebas de inserción 89
Figura 69. Rendimiento de Memoria B+-Tree Serializado Vs Virtual Machinery 90
Figura 70. Comparación de Memoria Libre 91
Figura 71. Comparación de los objetos utilizados 91
Figura 72. Comparación de creación de tipos de datos 91
Figura 73. B+-Tree 1,000,000 registro en memoria 93
Figura 74. Virtual Machinery 1,000,000 registros 93
Figura 75. Comparación de tamaño entre implementaciones 94

Índice de Tablas

Tabla 1. Algoritmos B-Tree 20
Tabla 2. B-Tree vs derivaciones 29
Tabla 3. Variaciones del B-Tree 31
Tabla 4. Comparación entre árboles 32
Tabla 5 Comparación analítica sobre B-Tree vs B+-Tree 33
Tabla 6 Algoritmos para un B+-Tree 35
Tabla 7 Requisitos mínimos para un dispositivo Android 36
Tabla 8 Mejoras al performance de Android [Hashimi 2010] 43
Tabla 9 Tipos primitivos en Android 45
Tabla 10. Comparación de métodos para implementar la persistencia 49
Tabla 11. FODA API de persistencia Java 50
Tabla 12. FODA Persistencia de Objetos (serialización) 50
Tabla 13. FODA Persistencia a bajo nivel 50
Tabla 14. Descripción de componentes 53
Tabla 15. Descripción clase Nodo 55
Tabla 16. Descripción clase Nivel 56
Tabla 17 Descripción clase NivelController 57
Tabla 18. Descripción clase ArbolMainController 58
Tabla 19. Descripción clase ListaDoble 58
Tabla 20 Layout del análisis de algoritmos 63
Tabla 21. Conjunto de datos de pruebas 84
Tabla 22. Comparación en operación de inserción (Unidades en milisegundos) 85
Tabla 23. Comparación en operación de búsqueda (Unidades en Milisegundos) 87
Tabla 24. Comparación en operación de eliminación 88
Tabla 25. Análisis de expansión de memoria B+-Tree Serializado 92
Tabla 26. Comparación de tamaños en disco 94

1. Capítulo Primero: Introducción

En 1972 el B-tree [Bayer 1972] revolucionó la manera en la que la información se almacenaba en discos magnéticos, dando pie a múltiples derivaciones del algoritmo, adaptables a problemas específicos con el fin de mantener una indexación balanceada, eficaz y escalable (porque minimiza el acceso a disco) para la búsqueda, inserción y eliminación de datos. La derivación más conocida del algoritmo B-tree en su optimización fue realizada por Knuth [Knuth 1973] y nombrada por Comer [Comer 1979], dicha implementación (B+-Tree) es la estructura fundamental para la indexación de grandes bases de datos y sistemas de archivos.

Por otro lado tenemos la evolución tecnológica, centrándonos específicamente en los dispositivos móviles que utilizan el sistema operativo de Google llamado Android, el que ha cambiado inclusive la máquina virtual de java para plataformas móviles *Java Micro Edition* (JME) por una propietaria llamada *Dalvik* [Guihot 2012]. La motivación surge del estudiar el comportamiento del B+-Tree fuera de los recursos ilimitados que tendría en un ordenador común y analizar la mejor forma de implementarlo en un dispositivo móvil con tecnología Android.

La problemática reside en proponer los mecanismos que permitan la implementación eficiente de un B+-Tree que supere las limitaciones técnicas de los dispositivos móviles en términos de: velocidad de procesamiento, consumo de energía, consumo de memoria RAM y ahorro de espacio físico.

El objetivo principal es pues, proveer un prototipo inicial del B+-Tree para Android, que tome en cuenta las limitaciones de la tecnología móvil para funcionar de forma eficiente. La implementación esta estructurada en una biblioteca de clases que permite la construcción de otras aplicaciones en Android, facilitando la implementación, escalabilidad, mantenimiento y persistencia.

El documento se desarrolla a lo largo de los siguientes capítulos:

- Capítulo 2: explica la historia y las razones de elección del B+-Tree para el presente trabajo;
- Capítulo 3: profundiza en la arquitectura de Android, así como sus limitaciones de Hardware y mejores prácticas en Software;
- Capítulo 4: se muestra el análisis de varios métodos empíricos de persistencia a fin de seleccionar el más óptimo;
- Capítulo 5: se enfoca al diseño e implementación de los diferentes B+-Tree que se realizaron para la plataforma móvil;
- Capítulo 6: analiza los resultados de la evaluación de nuestros prototipos y su comparación con una implementación comercial;
- Capítulo 7: se presentan conclusiones y trabajos a futuro para mejorar este estudio.

2. Capítulo Segundo: Los árboles

Debido a la necesidad para poder procesar información eficientemente, se han creado diversas estructuras que permitan indexar datos, es por ello que se da retrospectiva del árbol base creado por Bayer [Bayer 1972], el B-Tree, así como de algunas de sus derivaciones a fin de obtener un estado del arte que proporcione sus ventajas, limitaciones, aplicaciones, características y algoritmos, apelando al problema de la indexación.

La motivación del capítulo es conocer la historia del algoritmo base, la terminología y su funcionamiento, así como algunas de sus más importantes derivaciones.

Pero principalmente en este capítulo se encuentra la justificación de la elección del B+-Tree como algoritmo para indexación de datos en una plataforma móvil.

2.1 Conociendo el B-Tree

Cuando los dispositivos de almacenamiento vieron su luz los usuarios podían almacenar su información, consultarla desde los tan populares *Archivos*, para que esto pudiese ser posible la computadora tenía que leer desde el dispositivo de almacenamiento y poner la información en la memoria para poder procesarla y seguir interactuando con los *Archivos*, para hacerlo de manera eficiente se crearon varios algoritmos que manejaban la información de una manera inteligente acortando el tiempo de lectura y el tamaño del archivo, de ahí surge el algoritmo B-tree [Bayer 1972] [Bayer 1977] de la necesidad de organizar e indexar la información.

Rudolf Bayer y E. McCreight fueron los pioneros de la estructura B-Tree la cual se plasma en su artículo *Organization and Mantenience of Large Ordered Indexes* [Bayer 1972] en el que se habla sobre la organización de índices en medios físicos como un disco duro a manera de manipular información de éste último; permitiéndonos consultar, insertar, modificar y eliminar información por medio de la organización de los índices de manera ordenada, el documento se enfoca a la optimización de los accesos para obtener información con un enfoque meramente matemático denotando el énfasis de mejora con respecto al tiempo y al número de accesos para regresar la información solicitada, analizando su complejidad asintótica.

El objetivo es que las operaciones sobre la información y el medio físico sean proporcionales a:

$$log_k I$$

En donde I representa el tamaño del índice y k un número que representa el esquema más óptimo (o altura del árbol) permitiendo que la utilización del almacenamiento por nodo sea del 50% [Bayer 1972] [Comer 1979]. La manera en la que el algoritmo se desempeña es por medio de páginas que están indexadas de manera lógica y numéricamente ordenadas.

El B-Tree define un árbol como una colección de elementos pares (**índice, valor = (x,y)**) compuestos por una llave "**x"** con la propiedad de que sea única y un contenido o información asociada "**y",** la cual también puede ser un puntero hacia otra página dentro del árbol.

El objetivo del algoritmo se centra en economizar los recursos tanto de tiempo de acceso a la información como de espacio y organización, [Dongui 2003] menciona que tenemos a nuestra merced varios tipos de almacenamiento:

- **La memoria principal (caché) :** La cuál es muy rápida para poder acceder a la información pero es volátil y muy cara.
- **Los discos magnéticos:** Los cuales no son volátiles pero son mucho más lentos para manejar información, aquí es donde Bayer y McCreight se enfocan, tomemos en cuenta que su algoritmo B-Tree fue el pionero para obtener información de manera más rápida y organizada, el algoritmo data de 1972 donde la tecnología en cuanto al hardware era bastante precaria y una utilización de manera lógica era la única solución para manejar información de una manera más óptima.

A la par Bayer menciona a la caché como una unidad de almacenamiento muy pequeña y que es necesario guardar el volumen tremendo de índices que se procesan por la memoria principal en otra parte haciendo alusión a "fixed and moving head discs, drums and data cells" (Cabeceras de discos con movimiento, medios de almacenamiento y celdas de datos) [Bayer 1972]. Una vez almacenado de alguna manera en un archivo las bondades que brinda el B-Tree es el conjunto de operaciones tales como eliminar, insertar, modificar y consultar, todo de una manera económica y óptima, dependiendo de las necesidades y requerimientos de la aplicación podemos utilizar la memoria RAM, un medio físico o ambos para el almacenamiento de datos.

El B-Tree es una estructura de datos de tipo árbol, una de sus características principales es el estar balanceado, es decir, gracias a las operaciones básicas de inserción y eliminación, la estructura del árbol crece de manera controlada y equilibrada, teniendo la misma altura en todas sus ramas. Esto permite que la información pueda ser accedida con una complejidad constante *(O(log(n))),* permitiendo una indexación de los datos óptima.

Respecto al nombre B-Tree, no existen referencias que digan explícitamente el significado de la letra "B", sin embargo algunos autores [Dongui 2003] [Graefe 2011] [Twigg 2010] asumen que la "B" es por la palabra en el idioma inglés "Balanced" y Tree (Árbol) haciendo referencia a la estructura de datos que contamos en el ámbito computacional, en el que un consta de un conjunto finito de elementos, llamados nodos y un conjunto de líneas dirigidas llamadas ramas que conectan los nodos de forma balanceada.

En conjunto un B-Tree conforma un algoritmo que pretende ser la columna vertebral dinámica para el manejo de información, podemos medir la eficiencia del algoritmo con la Cota Superior Asintótica de orden logarítmica [Becker 1993] y decir que en el peor de los casos el árbol tendrá una complejidad en sus operaciones del orden de:

$$O(\log(n))$$

Por ejemplo en el artículo de Bayer se maneja un ejemplo de un árbol con una altura de 2 y un factor de ramificación de 1001 el cuál forzosamente necesita por lo menos 2 accesos a disco para buscar cualquiera de sus múltiples llaves (o nodos índice) , asumiendo que los accesos a disco son muy costosos, Comer [Comer 1979] denota que con las limitaciones de Hardware que se presentaban en esas épocas se buscaba optimizar los accesos a disco, ya que estos eran demasiado costoso, por lo que incesantemente empezaba a trabajar en derivaciones del algoritmo que permitieran optimizar dicho problema.

2.2 Estructura de un B-Tree

En cuanto a su estructura un B-tree puede contener un número variable de llaves e hijos en orden decreciente a la izquierda del nodo padre y creciente a la derecha del nodo padre respecto a la posición y valores del nodo hijo.

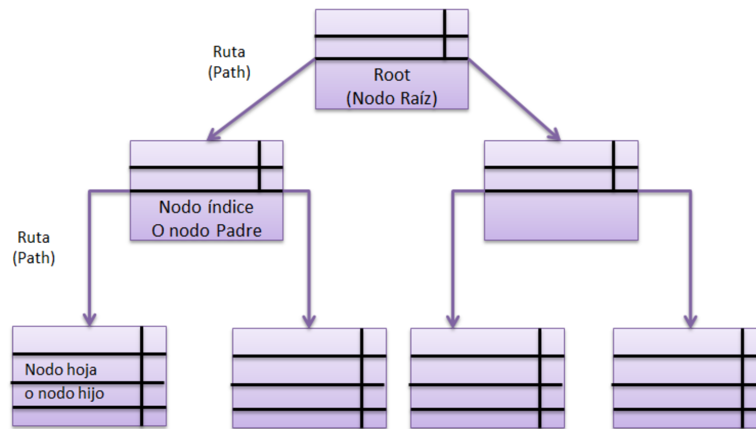


Figura 1. Estructura de un B-Tree

En la figura 1 se puede apreciar la estructura de cualquier árbol balanceado; compuesto de varios nodos distribuidos de manera ramificada, el nodo inicial se llama raíz. Y los nodos subsecuentes se llaman hijos o padres dependiendo si tienen derivaciones. El número máximo de hijos por nodo se llama *orden* del árbol., en donde cada *branch (rama)*, es navegable por medio de los apuntadores de los nodos, siguiendo una *path (ruta)* hasta encontrar los valores deseados consultando en cada nodo. El número máximo de accesos desde el nodo raíz hasta el nodo hoja se llama *profundidad* del árbol. En la figura 1 se ilustra un árbol de 4 hojas de orden 2 y profundidad 3. Un B-tree nos permite tener una organización por medio de llaves y valores acomodados de manera lógica y balanceada, en la figura 2 podemos ver un B-tree, identificando que bondades son muchas, siendo la principal: la búsqueda, ya que el árbol está organizado de manera inteligente para proveer *rutas (path)* acorde a la llave buscada en lugar de ser una búsqueda secuencial; otra de las bondades más sobresalientes es que el árbol es balanceado por medio de sus métodos (inserción y eliminación) los cuales controlan el tamaño y la estructura del árbol extendiéndola y

contrayéndola dinámicamente manteniendo las rutas cortas y haciendo el tiempo de búsqueda más eficiente [Bayer 1972] [Joyanes 2006] .

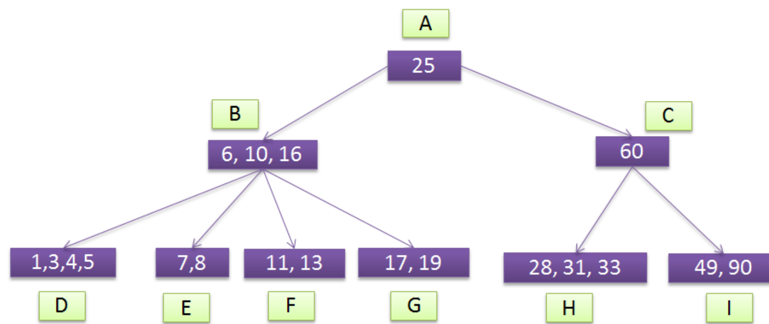


Figura 2. Ejemplo de un B-Tree

Para comprender la estructura de un B+-Tree [Dongui 2003], es importante conocer los siguientes puntos:

- Un nodo (puede ser un nodo hoja o un índice) *x* tiene un valor x.num en donde guarda el total de objetos de *x* de manera incremental. Para hacer referencia a un valor y para efectos representativos puede ser *x.key[i]* para su llave y *x.value[i]* para su valor en relación al índice i el cuál debe de apelar a la regla *1 <= i <= x.num* (es otras palabras *i* debe de ser menor al número total de objetos contenidos en nuestro nodo *k*).

- Cada nodo hoja debe contener la misma profundidad [Comer 1979].

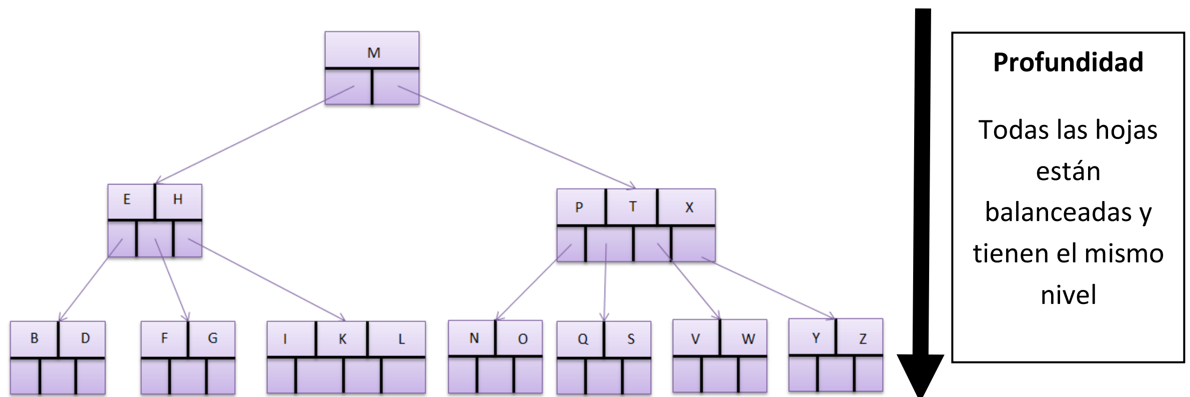


Figura 3. Balanceo en profundidad en un B-Tree

Como podemos observar en la figura 3, el árbol contiene una estructura que siempre mantiene el árbol balanceado, controlando la altura y haciendo que se un equilibrio natural debido a los algoritmos de inserción y eliminación [Bayer 1972][Bayer 1977][Dongui 2003].

- Un nodo index *x* además de guardar el número de nodos que contiene, también guarda un puntero en su posición *x.num + 1* que denota el ID de la página de su nodo hijo que es denotado por x.child[i] (para efectos de representación) el cuál obliga a un rango de valores (*x.key[i ¡ 1] , x.key[i]*), esto significa que en el subárbol cualquier llave debe ser mayor a *x.key[i ¡ 1]* y menor que *x.key[i]*. Por ejemplo en un subárbol referenciado por *x.child[1]* las llaves deben ser menores *a x.key[1]* y para un *x.child[2]* las llaves deben de estar entre *x.key[1]* y *x.key[2]* y así sucesivamente, como podemos apreciar en la figura 4.

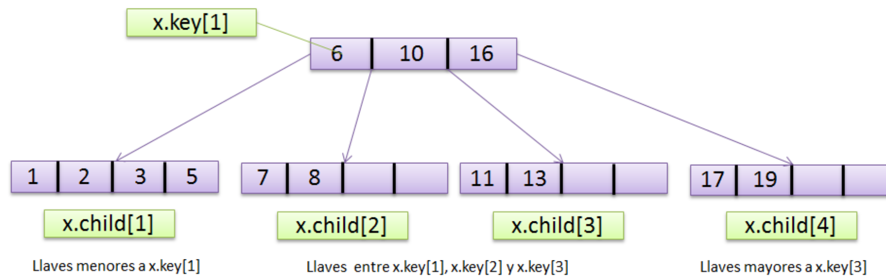


Figura 4. Lógica B+-Tree

- Cada nodo excepto el nodo raíz (root) debe de estar lleno por lo menos a la mitad, suponiendo que un nodo índice puede albergar 2B de hijos puntero (a su vez 2B-1 objetos), así cualquier nodo índice excepto el nodo raíz puede tener al menos B hijos punteros. Un nodo hoja puede albergar objetos, estos a su vez contienen datos. En otras palabras se va haciendo un *branching* o se va ramificando de menos a más generalizando un árbol que posee varios índices por cada nodo o página que apuntan hacia niveles inferiores en los que las hojas guardan los datos.

- Si el nodo raíz es un nodo índice debe tener por lo menos dos hijos.

Dada la naturaleza del algoritmo con sus métodos [Dongui 2003] encontramos que contiene funciones que ayudan a administrar los nodos en el árbol de una manera ordenada como se puede apreciar en la tabla 1:

Método	Descripción
Búsqueda	En esta función recursiva recibe como parámetros: el nivel del árbol en el que se revisará y la llave a buscar. Para empezar la búsqueda se toma como primer nivel el raíz y así ir navegando a través del árbol, comparando las llaves, hasta encontrar la llave y posteriormente consultar si tiene un nodo hoja que contenga el valor, el orden algorítmico es el siguiente.: • Leer la información del nivel (empezando cada búsqueda desde el nodo raíz) • Si el nodo contiene objetos de información compararlos con la llave y regresar el valor. • En caso de que no se encuentra regresamos la misma

	• función para hacer la recursión apelando al nivel más próximo a la llave. • En caso de que ya no tengamos nodos y de que no se encuentre el valor se regresa un valor nulo.
Inserción	La función de inserción se apoya de las funciones descritas a continuación, su trabajo es verificar dónde vamos a insertar el nuevo nodo y el reacomodar la estructura del árbol. Sus tareas en orden algorítmico son las siguientes: • Ubicar el nodo root (raíz) • Si el nodo raíz contiene espacio para contener el nuevo valor entonces se invoca al método insertarCuandoNoEstaLleno con los parámetros que son el nodo en el cuál se insertará, la llave y el valor. • Si el nodo raíz no contiene espacio se hace una reestructuración partiendo los objetos de información del nodo: - De todos los elementos de información del nodo se toma el de en medio (dado que están ordenados), llamándole *x*. - Se guardan todos los elementos de información que están a la izquierda de x en un nodo nuevo, llamándole *y*. - S e guardan todos los elementos de información que están a la derecha de x en un nodo nuevo, llamándole *z*. - Se reacomodan los punteros en caso de que sea una hoja índice y no una de datos. - Se hace el reacomodo elevando x hacia un nivel padre en donde los nodos *y,z* son sus nuevos hijos.
Eliminar	En un B-Tree se identifican 3 casos: Recibe como parámetros la llave que se desea eliminar y el nodo en el cual se encuentra o se cree encontrar la llave a eliminar. 1.- Cuando el nodo a eliminar es un nodo hoja que contiene objetos de información. Los pasos serían: • Eliminar el objeto con la llave que corresponda a la deseada para ser borrada. 2.- Cuando el nodo a eliminar es un nodo índice que contiene el objeto con la llave a ser eliminada. • Se localiza el nodo hijo en el que contenga el rango numérico apelando a la llave a eliminar. • Si éste nodo hijo esta exactamente a la mitad, entonces se debe de revisar la cantidad de objetos de información contiene el nodo de la izquierda y agregarle el objeto de información contenido en x, además debemos de equilibrar el árbol moviendo todos los objetos de *z* hacia *x*, destruyendo *z* y quedando nivelado.

	• En caso de que no esté exactamente a la mitad entonces mezclamos el nodo en el que se encuentra con el padre. • Se hace una recursión con el nodo actual y además de con la misma llave para que si se necesita reestructurar otra vez el árbol caiga en éste mismo caso o bien para que lo elimine sin complicaciones en el primer caso. 3.- Cuando el nodo a eliminar es un nodo en el cuál no contiene la llave que se desea eliminar. • Se ubica el nodo en el que se encuentra navegando en el árbol de manera recursiva hasta que se encuentre con el caso 1 o 2. • En caso de que los nodos *y* y *z* estén ½ llenos entonces los unimos y mandamos llamar recursivamente a la función eliminar.
insertarCuandoNoEstaLleno	Esta función nos sirve para poder insertar en el árbol cuando el nodo raíz no está lleno, sus tareas serían: • Verificar que si el nodo en el que se encuentra es un nodo hoja, de ser así se debe de insertar el nuevo objeto de información sin mayor complicación, manteniendo el orden incremental de los objetos de información. • En caso de que la hoja sea una índice debemos navegar hasta encontrar la hoja de datos que le correspondería; • En caso de que la hoja esté llena debemos de realizar el mismo procedimiento para separar en 3 nodos y reacomodar los punteros. • En caso de que no esté llena, simplemente insertamos en la hoja actual.
LocalizarNodo	Esta función sirve para encontrar el nodo en el cual debemos de insertar nuestro nuevo objeto de información.

Tabla 1. Algoritmos B-Tree

2.1 Algoritmos

2.1.1 Búsqueda

El algoritmo de búsqueda presentado es una función recursiva que recorre el árbol tomando en cuenta como parámetro de control la llave a buscar; Por ejemplo considere el siguiente árbol B+-Tree (figura 5):

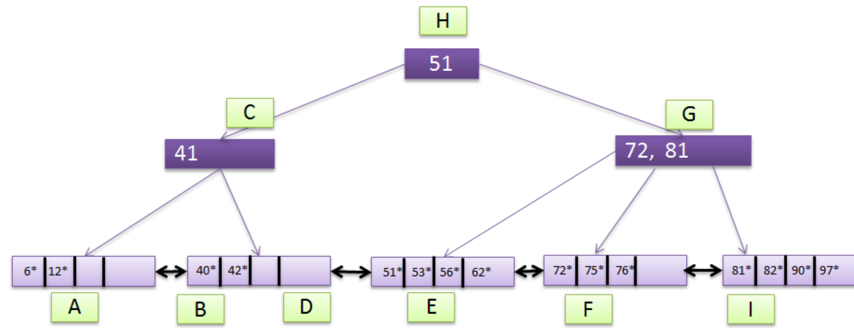


Figura 5 Búsqueda sencilla B+-Tree

Si se busca la llave o el índice 42 debemos de recorrer el árbol de manera comparativa y lógica, se empieza por la raíz y se evalúa que 51 es mayor a 42 por lo que se navega hacia la izquierda en donde tenemos otro nivel y se evalúa nuevamente, en este caso 42 es mayor 40 por lo que se navega hacia la derecha (En caso de que la comparación sea idéntica se navega hacia la derecha también) y se llega a un nivel hoja en el que se iteran todos los nodos del nivel para encontrar a la llave o índice buscado; de no ser encontrado se regresa un valor nulo.

Búsqueda (pagina_busqueda , llave_a_buscar)

```
1   nivel ← pagina_busqueda
2   if ( nivel es "indice" ) then
3       for each nodo in nivel do
4           Iterativamente se compara hasta encontrar el nodo con el índice
5           más cercano al parámetro llave a buscar y se regresa el nivel
6           actual
7           nivel ← nodo.siguiente_nivel
8           return  Busqueda(  nivel , llave_a_buscar )
9   else
10      Si es un nodo hoja
11       for each nodo in nivel do
12          if ( nodo.llave == llave_buscar )
13              return nodo
14  return null
```

<

En el algoritmo siempre se comienza por la raíz, tomando el inicio del árbol para encontrar primeramente el nivel hoja siempre y cuando el primer nivel no sea la raíz, esto se describe en las líneas 2 a la 5 en la que recorremos todos los routers del árbol para poder llegar hasta el nivel hoja, cuando esto llega se describe en las líneas 9 a la 14 el procedimiento para buscar dentro del nivel la llave deseada, de no existir entonces se regresa un valor nulo.

2.1.2 Búsqueda por rango

En el caso de que fuese una búsqueda por rango si se buscara desde el índice 42 hasta el 76, el B+-Tree está dotado con una lista doblemente enlazada en la que se realiza una búsqueda secuencial reduciendo la consulta por páginas pero incrementando el tiempo de búsqueda, ya que sabemos que el algoritmo secuencial es de los más pobres, una mejora notable sería implementar otro algoritmo de búsqueda sobre la lista enlazada; por ejemplo un *shaker sort*, éste evaluaría dentro del rango la llave solicitada.

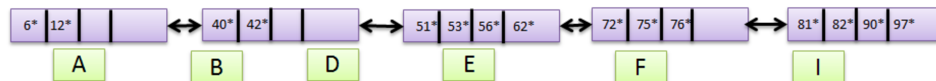


Figura 6 Búsqueda por rango B+-Tree

En la figura 6 podemos notar que el último nivel del árbol (nivel hoja) está representado por una lista doblemente enlazada en la todos los índices están ordenados de menor a mayor gracias a el balanceo del árbol, entonces basta con ir acumulando nodos desde que encontramos el índice inicial (40) hasta el índice final (76).

```
BusquedaPorRango ( índice_inicial , índice_final )

1   La variable lista se alimenta cada vez que se hace una inserción
2   Y se actualize cada vez que se hace una eliminación
3   nodos ← Array()
4     for each nodo in Lista do
5         if ( indice_inicial  >= nodo.llave AND indice_final <= nodo.llave ) then
6               nodos.add ( nodo )
7   return nodos
```

De esta manera se obtiene un arreglo de nodos dentro del rango deseable para la búsqueda, gracias a la lista doblemente ligada y la búsqueda funcional descrita entre las líneas 4 y 6 hasta llenar una colección de nodos que comprende el rango descrito por los parámetros.

2.1.3 Inserción

El algoritmo de inserción es vital para el B+-Tree ya que es el que realiza la estructura ordenada, lógica y correcta para la vida del árbol, su algoritmo es el que sigue:

Inserción (llave, valor)

```
1   Se realiza una búsqueda que determine en qué nivel será insertado
2   nivel ← getNivelAInsertar()
3   If ( nivel no está lleno) then
4      nivel.add( nuevo nodo (llave,valor)  )
5   else
6      while ( el nivel tenga un overflow ) do
7          iterativamente se reacomoda haciendo copy-up y/o push-up
8          nivel ←nuevo_padre //generado por el reacomodo
9   If (la raíz se parte) then
10     Se crea una nueva raíz con una sola llave y dos hijos
11  ListaDoble.add(nodo)
```

En las líneas 1 y 2 se realiza un proceso parecido a la búsqueda sencilla; donde se itera hasta llegar al nivel hoja correspondiente a la llave que se quiere insertar, una vez obtenido el nivel se procede a evaluar si está lleno apelando a la regla del ½ por nodo, si fuera el caso de que no estuviese lleno se inserta como lo demuestran las líneas 3 y 4, en caso de que el nivel a insertar esté lleno entonces se procede a un reacomodo del árbol, desde el último nivel del árbol (nivel hoja) haciendo un reacomodo iterativo para reajustar los niveles y los nodos haciendo copy-up y/o push-up como se describe en las ilustraciones 9 y 10 que a su vez se ejemplifica en las líneas 6 a la 8 en ésta última línea se apunta el nuevo nivel para seguir haciéndolo recursivo y con la ayuda del copy-up se re-direccionan los nuevos punteros hacia las ramas correspondientes del árbol mientras exista un desbordamiento (overflow) el cuál ocurre cuando un nivel tiene su capacidad máxima de nodos, finalmente en las líneas 9 y 10 se encuentra un caso único en que si la raíz hace un reacomodo entonces deberá de quedar con un nodo en el nivel raíz y dos niveles hijos.

En la figura 7 se realiza la inserción del número 6 en el pequeño árbol, la primera es que el B+-Tree está intacto, se desea insertar la llave 6, en la segunda parte de la figura 9, la llave 6 se inserta pero ocurre un problema el nodo ya no soporta más que 2 nodos por nivel entonces se realiza un push-up tomando de la nueva colección (5,6,7) la llave media, en nuestro caso el 6 para hacerlo padre, posteriormente se acomodan los punteros para los nuevos niveles que contienen a los nodos 5 y 7.

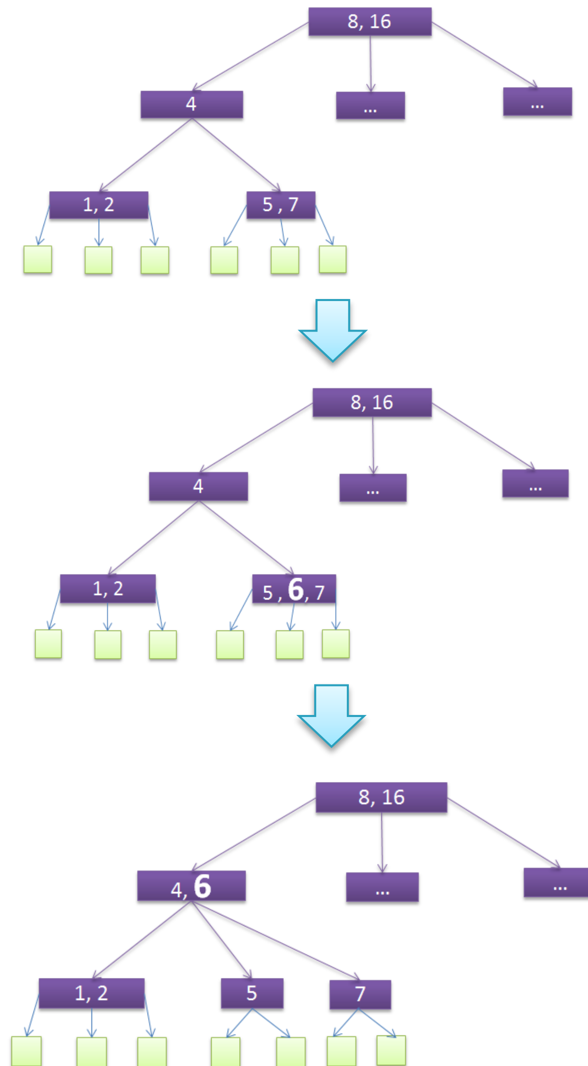


Figura 7. Inserción en un B+-Tree

2.1.4 Eliminación

El algoritmo de eliminación es necesario para el B+-Tree en caso de que se quiera tumbar algún nodo, éste algoritmo sirve además para re-estructurar la estructura del árbol cuando se da el caso. El algoritmo es como sigue:

Eliminación (llave)

```
1   Se realiza una búsqueda que determine en qué nivel será eliminado
2   nivel ← getNivelParaEliminar()
3   If ( nivel está > ½ lleno ) then
4      nivel.remove( nodo(llave) )
5   else
6      while ( el nivel tenga un underflow ) do
7        iterativamente:
8          if ( nivel_adyacente soporta un reacomodo ) then
9              Se redistribuye con el nivel_adyacente
10         else
11             Se unen nodos de nivel con el nivel_adyacente
12             nivel_adyacente.remove()
13            altura_arbol = altura_arbol – 1
14  listaDoble.remove(nodo(llave))
```

De manera similar a la inserción, se debe de encontrar la ruta en la cual se encuentra el nodo que vamos a eliminar, se puede dar el caso de que el nodo que se busca, no exista, pero en ese caso no pasa nada, solo un procesamiento sin meta cumplida. En las líneas del algoritmo de eliminación 1 y 2 se encuentra el posible nivel en el que se encuentra el nodo posteriormente en las líneas 3 y 4 se checa el nivel para ver si soporta una eliminación (es decir que el nodo quede con mínimo ½ nodos) entonces se elimina el nodo y no existe mayor problema. En caso de que no pase el proceso de reacomodo, se comienza desde la línea 6 a la 13 iterativamente, mientras exista underflow se checa si se puede redistribuir con su nodo adyacente en caso contrario lo mezclamos con su nodo adyacente y eliminamos el nivel que sobra y finalmente disminuye la altura del árbol.

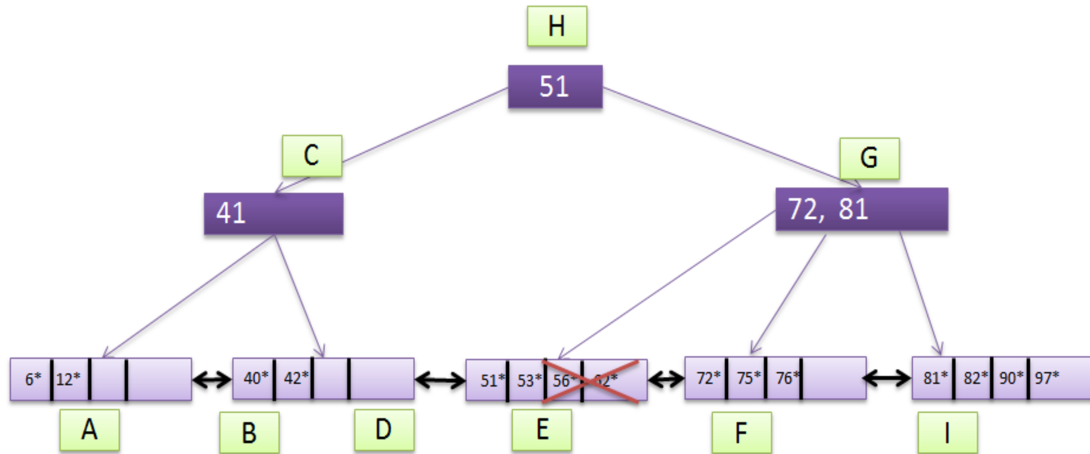


Figura 8. Eliminación sencilla

En la figura 8 se muestra una eliminación que no tiene mayor complicación removiendo del árbol las llave 62, como no alteran la estructura del árbol y el nodo D sigue conservando los requisitos entonces sólo se remueven, apelando a las líneas del algoritmo 3 y 4.

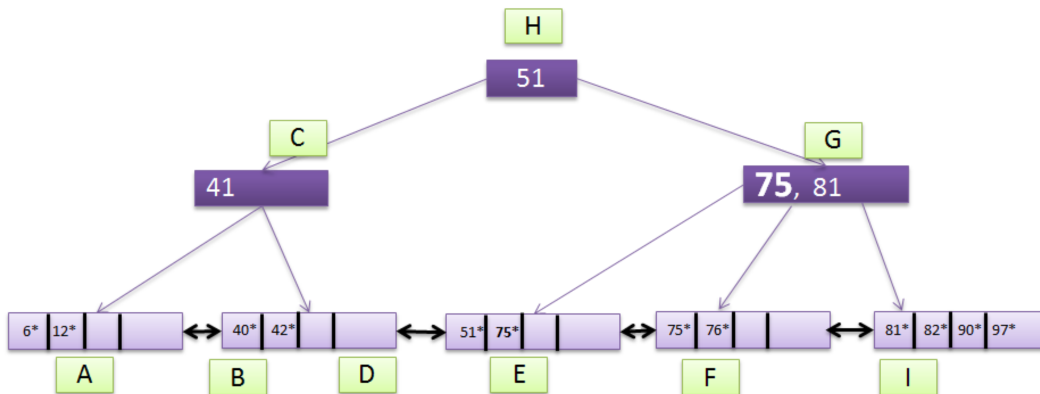


Figura 9. Eliminación con re-distribución

Ahora, si se toma en cuenta la figura 8 y queremos remover la llave 53 observamos que se hace una distribución en la que el 72 pasa al nodo D debido a que no cumple los requisitos de estar ½ lleno y el router se cambia al valor siguiente 75 para preservar el orden y el flujo del árbol, tal como se puede apreciar en la figura 9, apelando a las líneas del algoritmo 6-9.

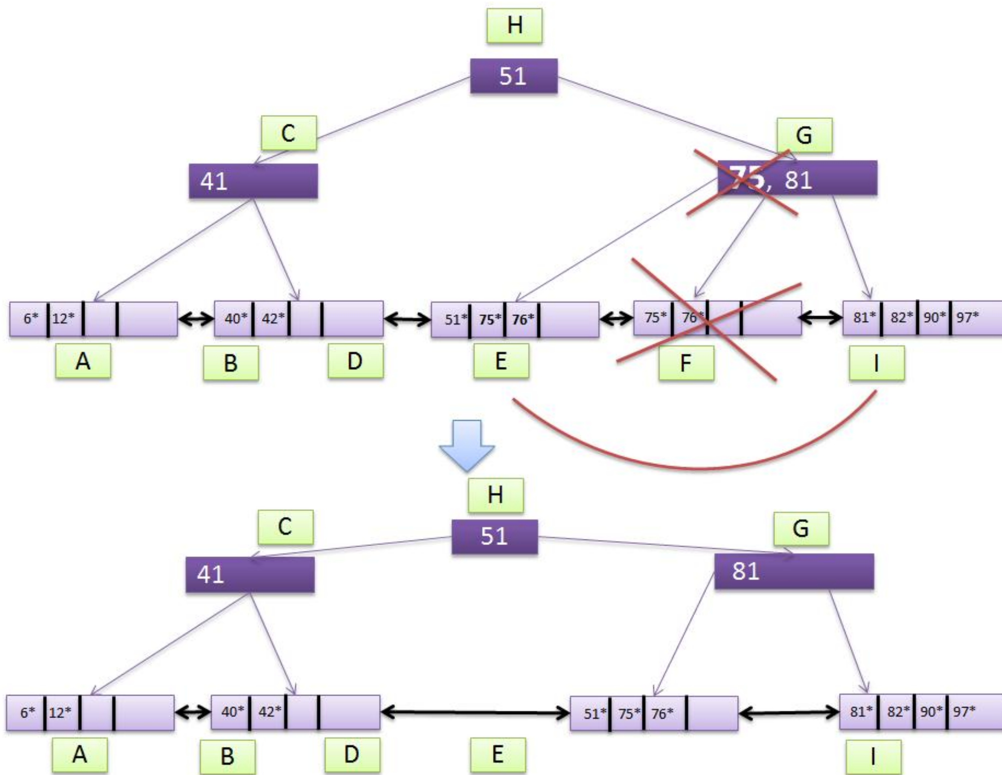


Figura 10. Eliminación B+-Tree Mezclando niveles

En la figura 10 se supone que se elimina la llave o índice 72, lo cual da un problema porque el nodo D no alcanza a ser distribuido y ocurre un underflow, por lo que se tiene que hacer una mezcla con todos los elementos del nodo *E*, eliminar el nodo adyacente (*E*) y reacomodar los routers, en caso de que el nivel eliminado fuera el único disminuiría la altura del árbol. Apelando a las líneas 6,11 y 12 del algoritmo.

2.2 Variaciones del B-Tree

Knuth libera la derivación llamada B*-Tree [Knuth 1973] en la que en lugar de que los nodos estuvieran ½ llenos, el comprobó que el algoritmo presentaba casos de mejora si se manejaban nodos a 2/3 de su capacidad total, además en su inserción emplea un esquema de distribución que retrasa el reacomodo hasta que 2 nodos estén completamente llenos, después los 2 nodos son divididos en 3 cada uno con 2/3 partes de su capacidad total garantizando que por lo menos el 66% de la eficiencia en el reacomodo incrementando así la utilización del espacio físico, mayor rendimiento en el tiempo y disminuyendo la altura del árbol. Sobre los B*-Tree surgieron varias variaciones, otra es la llamada Prefix B+-Trees [Bayer 1977] en la que la técnica consiste en elegir un prefijo y en base al prefijo sortear las llaves para implementar las búsquedas y las inserciones de manera más rápida, el problema es la forma en el que implementamos el prefijo afecta el comportamiento del árbol y disminuye el desempeño. Otra variación son los Virtual Trees que fueron diseñados para hardware especial que incrementando la velocidad, la protección de la memoria aísla otros usuarios (para tener un reservado para B-tree), almacenando las partes más consultadas del árbol en memoria, como un caché. Bayer [Comer 1979], propone otra variación llamada Binary B-Tree el cual eficiente la pérdida de espacio de los nodos que están a ½ de su capacidad implementando un espacio más en el nodo que representa un puntero a ser hermano o a irse para abajo, obteniendo una distribución estructural más inteligente del árbol y salvando espacio.

Son muchas las variaciones del B-Tree y también existen variaciones de las variaciones, por ejemplo otra variación interesante de los B-Tree que impacta en la actualidad son los R-Tree [Guttman 1984]. La ventaja principal y la más potencial de ésta variación es que a diferencia de los B-tree que manejan datos alfanuméricos como los enteros, los tipo char y las cadenas, un R-tree puede manejar datos geométricos como puntos, segmentos de línea, superficies, volúmenes, hyper-volúmenes en diferentes capas dimensionales con respecto al espacio, aplicándose al manejo de la multimedia y al apoyo de los sistemas geográficos, simplemente de éste derivado existen más de 20 diferentes implementaciones, de las cuales algunos mejoran el algoritmo y/u otras lo perfilan para propósitos específicos como el 3D R-tree para herramientas CAD [Manolopoulos 2010].

Los B-tree funge pues, como el algoritmo principal del cual se desglosan muchísimas implementaciones para el manejo de sistemas a sus medios físicos, permitiendo a bases de datos manipular información por medio de la indexación. Un ejemplo de la implementación del B-Tree está implícito en la estructura que utiliza Mysql, InnoDB [Mysql 2012] .

Dadas las ventajas anteriores un B-tree también funge como una columna vertebral para una base de datos, otorgando una funcionalidad como un motor de almacenamiento, en la propia experiencia que se tiene como desarrollador, se ha utilizado InnoDB y MySAM (aunque nunca se vió a detalle las diferencias que estos contenían). Complementando la comparación de [Blanco 2007], en la tabla 2, se agregan las ventajas y desventajas que se tiene con respecto al B-Tree,

MySAM e InnoDB, a fin de destacar la importancia del B-Tree en su participación sobre base de datos.

Algoritmo	Ventajas	Desventajas
ISAM	Fácil implementación	No es eficiente en grandes volúmenes de datos
Hash	Se puede tener muy buen tiempo de respuesta en muchos casos	Difícil implementación y no existen métricas del comportamiento "trie"
B-tree	Fácil implementación y extremadamente rápido Buena compresión de datos	Sacrifica un poco de espacio por repetición de algunos elementos del árbol (actualmente es despreciable debido a las variaciones)- Utiliza medios de almacenamiento alternativos
MySAM	Buena compresión de datos	Utiliza mucho el caché Almacena como caen los datos Deja que el sistema haga el caché de las lecturas y escrituras No soporta transacciones
InnoDB	Almacena acorde a llave primaria Aprovecha logs para no repasar todos los índices	No dispone de una buena compresión Ocupa mucho más espacio en memoria y disco

Tabla 2. B-Tree vs derivaciones

En el sitio web de MySQL nos dice que InnoDB está basado en B-tree [Mysql 2012] y utiliza el mismo algoritmo de compresión con un método de indexado llamado "index-organized table" (Tabla de organización de índices el cuál apela al principio de Bayer [Bayer 1972]) en el cuál cada fila en el índice contiene valores de la llave primaria y de todas las columnas en su tabla. Es interesante saber además que los índices secundarios en InnoDB son B-trees independientes que contienen legítimamente la tupla de valores (la llave y un puntero a una fila en un índice agrupado).

Desde que Bayer en el año de realizó el B-Tree han surgido infinidad de variaciones [Tao 2010] y especificaciones que toman como base el algoritmo inicial para solventar problemas informáticos de diversas índoles [Manolopoulos 2010], una recopilación de diversas variaciones, se muestra en la siguiente tabla, hay que tomar en cuenta que la tabla fue complementada para dar algunas otras variaciones que dejaron fuera:

Año	Variación	Autores
1962	AVL tree	G.M Adelson-Velskii
1972	Red-black tree	Bayer
1979	B+-Tree	Knuth

1985	Splay tree	Daniel Dominic Sleator, Robert Endre Tarjan
1994	Hilbert R-tree	Kamel, Faloutsos
1994	R-link	Ng, Kameda
1994	TV-tree	Lin, Jagadish, Faloutsos
1996	QR-tree	Manolopoulos, Nardelli, Papadopoulos, Proietti
1996	SS-tree	White, Jain
1996	VAMSplit R-tree	White, Jain
1996	X-tree	Berchtold, Keim, Kriegel
1996	3D R-tree	Theodoridis, Vazirgiannis, Sellis
1997	Cubtree	Roussopoulos, Kotidis
1997	Linear Node	Splitting Ang, Tan
1997	S-tree	Aggrawal, Wolf, Wu, Epelman
1997	SR-tree	Katayama, Satoh [108]
1997	STR R-tree	Leutenegger, Edgington, Lopez
1998	Bitemporal R-tree	Kumar, Tsotras, Faloutsos
1998	HR-tree	Nascimento, Silva
1998	Optimal Node Splitting	Garcia, Lopez, Leutenegger
1998	R¤*a*-tree	Juergens, Lenz
1998	STLT	Chen, Choubey, Rundensteiner
1998	TGS	Garcia, Lopez, Leutenegger
1999	GBI	Choubey, Chen, Rundensteiner
1999	RST-tree	Saltenis, Jensen
1999	2+3 R-tree	Nascimento, Silva, Theodoridis
2000	Branch	Grafting Schrek, Chen
2000	Bitmap R-tree	Ang, Tan
2000	TB-tree	Pfoser, Jensen, Theodoridis
2000	TPR-tree	Saltenis, Jensen, Leutenegger, Lopez
2001	aR-tree	Papadias, Kanlis, Zhang, Tao
2001	Box-tree	Agarwal, deBerg, Gudmundsson, Hammar, Haverkort
2001	Compact R-tree	Huang, Lin, Lin
2001	CR-tree	Kim, Cha, Kwon
2001	Efficient HR-tree	Tao, Papadias
2001	MV3R-tree	Tao, Papadias [

2001	PPR-tree	Kollios, Tsotras, Gunopulos, Delis, Hadjieleftheriou
2001	RS-tree	Park, Heu, Kim
2001	SOM-based R-tree	Oh, Feng, Kaneko, Makinouchi
2001	STAR-tree	Procopiuc, Agarwal, Har-Peled,
2002	aP-tree	Tao, Papadias, Zhang,
2002	Buffer R-tree	Arge, Hinrichs, Vahrenhold, Vitter,
2002	cR-tree	Brakatsoulas, Pfoser, Theodoridis,
2002	DR-tree	Lee, Chung,
2002	HMM R-tree	Jin, Jagadish,
2002	Lazy Update R-tree	Kwon, Lee, Lee,
2002	Low Stabbing Number	deBerg, Hammar, Overmars, Gudmundsson,
2002	VCI R-tree	Prabhakar, Xia, Kalashnikov, Aref, Hambrusch,
2003	FNR-tree	Frentzos,
2003	LR-tree	Bozanis, Nanopoulos, Manolopoulos,
2003	OMT R-tree	Lee, Lee,
2003	Partitioned R-tree	Bozanis, Nanopoulos, Manolopoulos
2003	Q+R-tree	Xia, Prabhakar,
2003	Seeded Clustering	Lee, Moon, Lee.

Tabla 3. Variaciones del B-Tree

Contemplando las diversas implementaciones derivadas del B-Tree se identifican 5 principales: AVL-Tree [Bayer 1977], B+-Tree [Comer 1979] , Red-black tree [Oracle 2012] R-Tree [Guttman 1984] y Splay trees [MagickCore 2012], las demás son variaciones de las variaciones del B-Tree o bien no son/fueron tan exitosas. Ahora se presenta una tabla que compara aspectos de las diferentes variaciones:

Nota: El algoritmo B-Tree no es considerado ya que el B+-Tree es una mejora inmediata al tener los datos en los nodos hoja, la diversificación de los nodos a partir de los 2/3 de la hoja llena y la búsqueda por rango implementando una lista enlazada.

Antes de empezar el análisis tipo benchmarking es importante recalcar que cada uno de los árboles, son utilizados para diferentes propósitos, por lo que no se trata de hacer una comparación entre ellos, sino un análisis de ver cuál es el más factible para ser transportado a una plataforma Android. Es importante conocer que todos ellos poseen la misma interface: insertar, eliminar, modificar y consultar, aunque sus usos y lugares de ejecución varíen entre sí. Por ejemplo el R-Tree, B+-Tree y AVL Tree son utilizados para guardar información en disco, en cambio

el Red-black tree y Splay tree son más utilizados en memoria, no por ello significa que no se puede implementar para escribir en disco, aunque no están diseñados para esos fines.

Característica	B+-Tree	Red-black tee	R-Tree	AVL-Tree	Splay tree
Complejidad búsqueda, inserción y eliminación.	O(logn)	O(logn)	O(logn)	O(logn)	O(logn)
Comúnmente utilizado para almacenamiento ROM	ok		ok	ok	
Comúnmente utilizado en memoria		ok			ok
Con ramas y nodos balanceados	ok		ok		
Con ramas y nodos desbalanceados		ok		ok	ok
Mínimos accesos a disco/memoria debido al balanceo de las hojas.	ok		ok		
Buen comportamiento con un conjunto de datos pequeños.	ok	ok	ok	ok	ok
Buen comportamiento con un conjunto de datos grandes	ok		ok		

Tabla 4. Comparación entre árboles

Podemos observar en la tabla 4 que los árboles más óptimos para trabajar tanto en disco como en memoria son el B+-Tree y el R-Tree, los árboles AVL no se toman en cuenta ya que son superados con facilidad por un B-Tree. El Red-black tree es rápido en memoria pero muy lento en disco, ya que está enfocado a funcionar como un árbol binario para hacer operaciones como ordenamientos o búsquedas en conjuntos de datos pequeños, al igual que el Splay Tree, con la diferencia de que éste último tiene una gran desventaja, dependiendo de las operaciones que se realicen sobre el, puede llegar a un estado lineal, lo que obligaría a realizar una búsqueda secuencial que sería bastante impráctica, lenta y tediosa. Mientras tanto el B+-Tree al ser un árbol balanceado mantiene su complejidad asintótica.

Por tanto los algoritmos que podemos utilizar con más eficiencia tanto en memoria como en disco son el B+-tree y el R-tree, éste último es descartado debido a que se utiliza exclusivamente para manejar información y datos multidimensionales tales como coordenadas geográficas, rectángulos o polígonos.

Entonces pues, utilizaremos el algoritmo B+-tree para la implementación en Android al ser el más eficiente tanto en memoria/disco y al tener propiedades de balanceo en sus ramas y hojas, sin contar las características que lo hacen poderoso tales como: la reducción de espacio por hoja (2/3 full) y la lista enlazada que manejan las hojas para hacer búsquedas por rango.

No obstante, no todo es tan perfecto, ya que aún y cuando es el algoritmo más óptimo, también tiene deficiencias, ya que se gasta más tiempo en insertar los nodos en la lista enlazada y al mantenerla en memoria y/o al levantarla de la misma, y los accesos a disco varían respecto a la altura del árbol.

2.3 El B+-Tree

Acorde a la sección, se elige el algoritmo B+-Tree por las bondades que ofrece con respecto a las demás variaciones, el B+-Tree denota la variación más conocida del B-Tree, misma que fue propuesta ideológicamente por Knuth y nombrada finalmente por Comer [Comer 1979] fue la variación B+-Tree, a continuación se hace una comparación en la tabla 5, entre el B-Tree [Bayer 1972] y el B+-Tree, exponiendo las ventajas que el algoritmo B+-Tree tiene sobre el B-Tree.[Dongui 2003] [Graefe 2001] :

Caso	B-Tree	B+-Tree
Guardado de los objetos de información	Se pueden guardar en cualquier nivel que contenga hojas para datos u hojas índices. Haciendo el algoritmo más lento, ya que al momento de hacer la navegación para llegar a la hoja buscada tiene más carga que leer por cada nodo que contenga información, recordemos que el algoritmo de búsqueda se utiliza tanto para insertar como para eliminar.	Todos los objetos de información se encuentran en el último nivel del árbol. Lo que eficiente el algoritmo ya que facilita la navegación con nodos índices hasta llegar a los nodo hoja que están en el último nivel. Ya no existe preocupación del por el cómo quedarán los sub-árboles, ya que todos los datos se encuentran en el último nivel.
Búsqueda	La búsqueda solo retorna el resultado de la llave que buscamos. En algunos casos puede ser más eficiente ya que al tener menos nodos puede ser más rápido el regresar el valor buscado antes de llegar al último nivel.	Maneja una lista doble en la que se guardan los elementos, facilitando la búsqueda por rangos, es decir se puede buscar los elementos desde la llave 87 hasta la 92, y no solo obtener un solo resultado, disminuyendo el acceso a discos y/o a memoria en algunos casos.
Nodos índices	En los nodos índices se pueden guardar valores. De alguna manera esto ahorra espacio tanto en memoria como en disco	Un nodo índice solo actúa como un router para decidir hacia dónde ir. Al tener muchos nodos índices se desperdicia mucho espacio, ya sea en memoria o en disco.

Tabla 5 Comparación analítica sobre B-Tree vs B+-Tree

El B+-Tree hace mejoras significativas al algoritmo del B-tree haciéndolo más simple y por consecuencia de más fácil implementación, para manejar los desbordes (overflows) se manejan técnicas llamadas Copy-up y Push-up que se ejecutan cuando un nodo llega a su límite o presenta un “Overflow” (Desbordamiento), maneja un reacomodo parecido al del B-tree pero en la capa de datos:

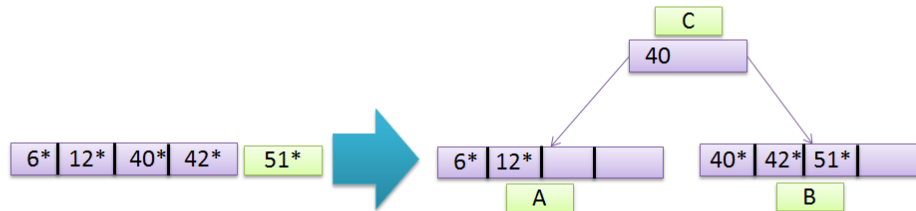


Figura 11. Copy-up

Exclusivamente cuando existe un desbordamiento al nivel de los datos y se desea insertar un elemento en un bloque que ya está lleno, por ejemplo el 51, el bloque es desbordado, tomando la llave intermedia y copiándola al nodo índice padre (C), los nodos son repartidos entre un nuevo bloque y el que ya existía (A y B). Por otro lado en caso de que el desbordamiento ocurriera a nivel de nodos índice (figura 5), se realiza una operación llamada push-up, en la que la llave intermedia es movida hacia el nodo padre (H) y removiéndola del nodo en el que se encontraba.

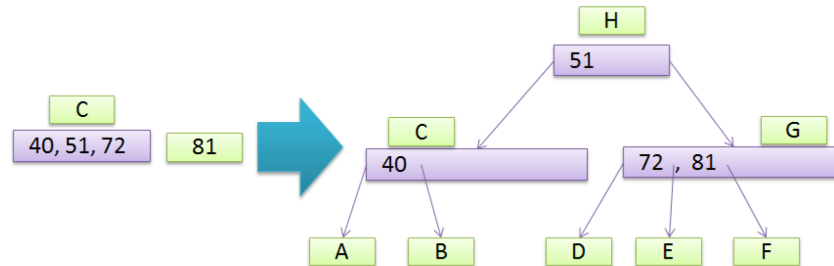


Figura 12. Push up

Ejemplificando, en la figura 6 se inserta la llave 81 y se hace un push up a la llave 51, del mismo modo si se insertase una llave 60, el nodo G desbordaría y tendríamos un push up del valor 72 al nodo H.

En cuanto a la descripción de los algoritmos podemos notar que tienen menor complejidad que los del B-tree ya que manejan los siguientes métodos descritos por Comer [Comer 1979] y complementados por [Dongui 2003] en a tabla 6:

Método	Descripción
Búsqueda	Parametrizada con la página de búsqueda y con la llave(s) a buscar ya que puede manejar el rango de llaves para regresar múltiples valores. • Se hace una búsqueda lógica comparando la llave desde la raíz hasta el último nivel que son las hojas diferenciando entre nivel hoja e índice (Si la altura del árbol es 1 la raíz es el nivel de hoja) • Se obtiene la primer llave (si existe) y se regresa el valor • Si la búsqueda pide un rango, se consultan los valores haciendo una búsqueda secuencial en la lista enlazada.
Inserción	Parametrizada con el nivel la llave y el valor. Tomando en cuenta de que la llave no debe de existir en el árbol.

	• Se hace una búsqueda de la llave que regreseel nodo hoja donde debemos de insertar el objeto de información. • Se inserta el objeto de información. • Si el nodo donde se insertó tiene overflow entonces se hace el proceso de copy-up y/o de push-up • Si el nivel es 1 se crea un nuevo nodo índice como root y se separa en dos nodos índices.
Eliminación	• Se hace una búsqueda de la llave que nos regrese el nodo hoja donde debemos de eliminar el objeto de información. • Eliminamos el objeto si es que se encuentra. • Comprobamos la altura del árbol si es 1 entonces todo es correcto porque lo quitamos del nodo raíz. • Si la altura no es 1, entonces mientras el nodo retrocede en el árbol haciendo un underflow se reacomodan los routers y se hacen reacomodaciones en el árbol y dejarlo balanceado solo si un nodo tiene más entradas que las mínimas ocupadas. • En caso de que el nodo no tenga las entradas mínimas se mezclan los objetos de información del nodo con el contiguo y se decrementa la altura del árbol.

Tabla 6 Algoritmos para un B+-Tree

Queda claro que el B+-Tree es un algoritmo que es mucho más simple de implementar y ofrece múltiples ventajas sobre el B+-Tree, optimizando las operaciones de inserción y eliminación para dejar a los nodos hoja (los que contienen los datos) al final del árbol, además de hacer un balanceo más diversificado al mantener el 2/3 de capacidad en cada página del árbol.

El B+-Tree entonces, ha demostrado ser la derivación más óptima para ser implementada en dispositivos móviles, con mira a resolver un problema de indexación tomando en cuenta las limitaciones de la tecnología móvil.

3. Capitulo Tercero: Arquitectura y mejores prácticas en Android

El presente capítulo muestra la arquitectura de la plataforma Android con el objetivo de ver cómo se ejecuta una aplicación, y su relación con el hardware del dispositivo host: la capacidad del procesador; las cualidades de la memoria RAM; y las características del medio de almacenamiento.

La motivación principal es la de conocer a fondo la estructura de una plataforma móvil para crear un algoritmo eficiente que pueda ser transportable, flexible y robusto para manejar registros por medio de un B+-tree.

La estructura del capítulo se divide en cuatro secciones: la primera habla de los requisitos mínimos de hardware para poder utilizar Android en cualquier dispositivo móvil; la segunda describe la arquitectura de la plataforma Android; mientras que la tercera dice cómo es que el software se ejecuta apelando a la arquitectura antes mencionada y las consideraciones a tomar en cuenta para hacer mas eficiente una aplicación; finalmente la cuarta nos describe como hacer eficientes las aplicaciones en la plataforma Android.

3.1 Requisitos mínimos en la plataforma móvil Android

Partimos de la premisa en la que Android es un sistema operativo para plataformas móviles basado en Linux para ofrecer servicios tales como seguridad, administración de memoria, administración de procesos [Kumar 2008], y un modelo de red. Por otra parte las aplicaciones están programadas en el lenguaje Java, para ello se tiene que tener una máquina virtual que sea capaz de compilar e interpretar las líneas de código necesarias para consolidar así una aplicación funcional. A sabiendas que Android tiene un kernel de Linux versión 2.6.29, la arquitectura del dispositivo fue pensada, para ambientes con restricciones de memoria, de poder de procesamiento y de almacenaje, por lo cuál los requisitos mínimos [Ehringer 2010] para un dispositivo se pueden apreciar en la tabla 7:

Requisito	Mínimo requerido
Chipset	ARM-Based
Memory	128 MB RAM; 256 MB Flash External [Ehringer 2010] 64 MB RAM (40 System – 20Apps) [Hashimi 2010] **[AndroidConferences 2012]**
Storage	Mini o Micro SD
Primary Display	QVGA TFT LCD de 16 bits
Camera	2 MP CMOS
USB	Standard mini-B USB interface

Tabla 7 Requisitos mínimos para un dispositivo Android

Es importante conocer que cada día la tecnología avanza y que en la actualidad existen dispositivos que fácilmente rebasan las características mínimas que aquí se presentan, pero aún así no son comparables con los que se disponen, por ejemplo, en una workstation. Al tener un procesador de velocidad limitada, una memoria RAM limitada, el no tener compartición con el disco (SWAP), etcétera. Con esto podemos gestionar un algoritmo eficiente que se pueda adecuar lo más cerca posible a los requerimientos mínimos para su óptima ejecución, o bien que explote los pocos recursos disponibles para poder alcanzar la cúspide de su funcionalidad.

3.2 Arquitectura Android

Resulta indispensable el conocer la arquitectura de Android para realizar cualquier aplicación o modificación a sus partes, en la figura siguiente se muestra de manera abstracta y gráfica la arquitectura, para dar una idea general de las partes que el algoritmo B+-Tree impactará para alcanzar su eficiencia.

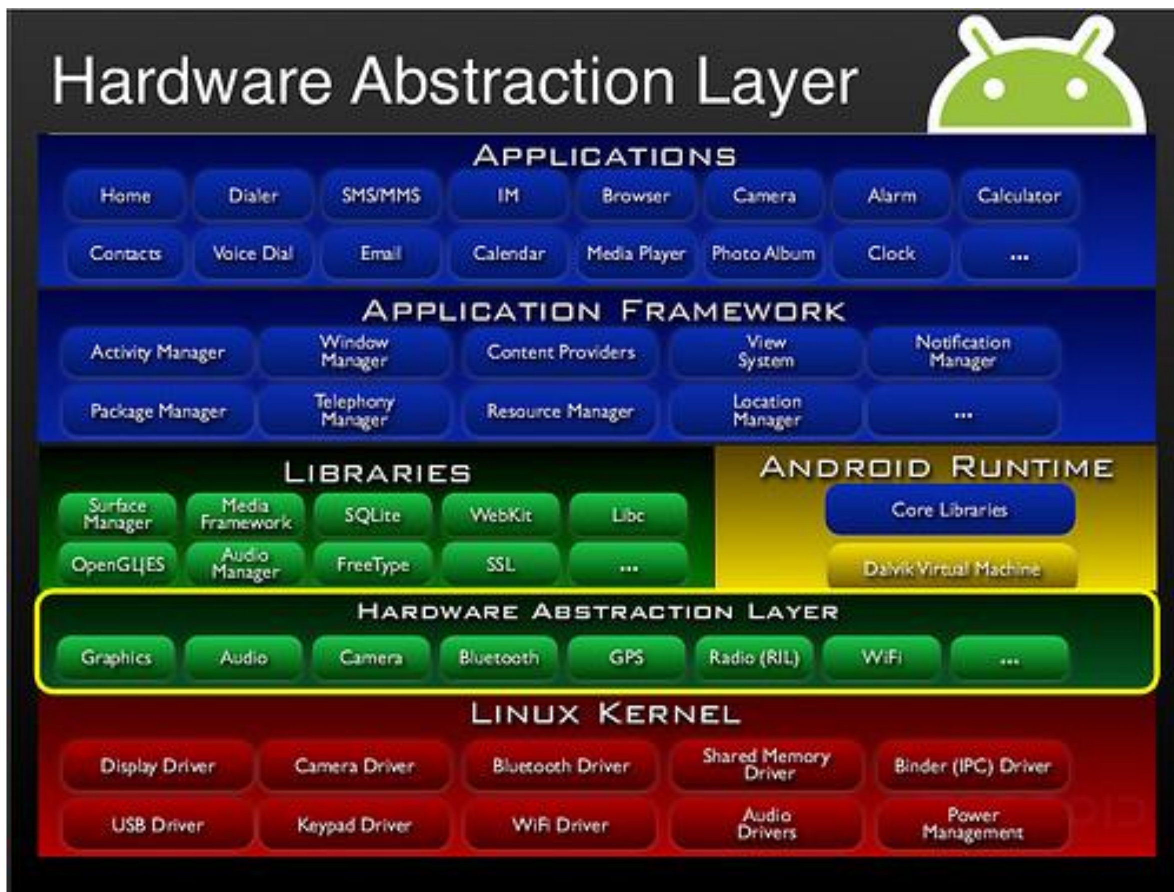


Figura 13. Arquitectura de la plataforma Android [Ehringer 2010]

Podemos contemplar en la figura 13, que en la capa de bajo nivel cohabita el kernel de Linux, en el cual se encuentran los drivers de los componentes que soporta el dispositivo Android, por encima se tiene una capa de abstracción del hardware que es la que se encarga de manipular a bajo nivel las instrucciones digeridas por las capas superiores. Android tiene un conjunto de librerías pre-cargadas que pueden ser ocupadas por todas las aplicaciones a fin de traducir las

instrucciones a bajo nivel y poder así ser interpretadas, en ésta misma capa habita la máquina virtual Dalvik: ésta máquina virtual fue diseñada especialmente para dispositivos móviles. La capa superior a esta es un framework comunitario para las aplicaciones en las que pueden acceder al manejo de las ventanas, a los administradores de actividades (Zygote[2]) [Guihot 2012], entre otras, para así tener una amplitud en su funcionalidad. Siendo la última capa la de aplicaciones que utiliza el engine descrito con anterioridad.

3.3 Ejecución del Software y consideraciones sobre la plataforma Android

Es importante el conocer el como se ejecuta el software sobre la plataforma Android, en este apartado conoceremos la máquina virtual que permite la ejecución de las aplicaciones; después indagaremos en los tipos de memoria que maneja Android; se finaliza describiendo la ejecución de una aplicación en el sistema operativo portátil.

3.3.1 Máquina Virtual Dalvik

Al ser Java el lenguaje para el desarrollo de aplicaciones para Android, se tiene que tener por fuerza una máquina virtual que lo compile y ejecute, antes de ahondar de lleno en la máquina Dalvik conviene hacer una retrospectiva hacia su origen.

Java al ser un lenguaje con el lema “write once, run anywhere” ha sido reconocido por su portabilidad a diferentes ambientes y dispositivos, tan es así que se tienen varias instancias tales como: la JEE (Java Enterprise Edition)[Hashimi 2010] la cual es la más completa con funcionalidades específicas para servidores; la JSE (Java Standard Edition) [Hashimi 2010] que es comúnmente utilizada en equipos de escritorios; la JME (Java Micro Edition) [Hashimi 2010] que está más fragmentada que las anteriores y tiene modificaciones importantes para ser soportada por diferentes dispositivos, Así pues Google decide tener su propia máquina virtual denominada Dalvik [Ehringer 2010] apelando a una implementación limitada pero funcional de las librerías principales del Java.

Dalvik compila y ejecuta de manera diferente los archivos java, en lugar de obtener un archivo *.class* genera archivos *.dex*, los archivos dex apelan a las limitaciones que comparten los dispositivos móviles con sistema operativo Android, ya que están pensados inteligentemente para que puedan ser más eficientes que los archivos *.class*, conteniendo un nivel de compresión mayor, mejor flujo de los datos y diseñados estratégicamente para manejar las especificaciones del lenguaje java, dentro del contexto de la máquina Dalvik, la figura 14 podemos ver las diferencias en la estructura del archivo compilado:

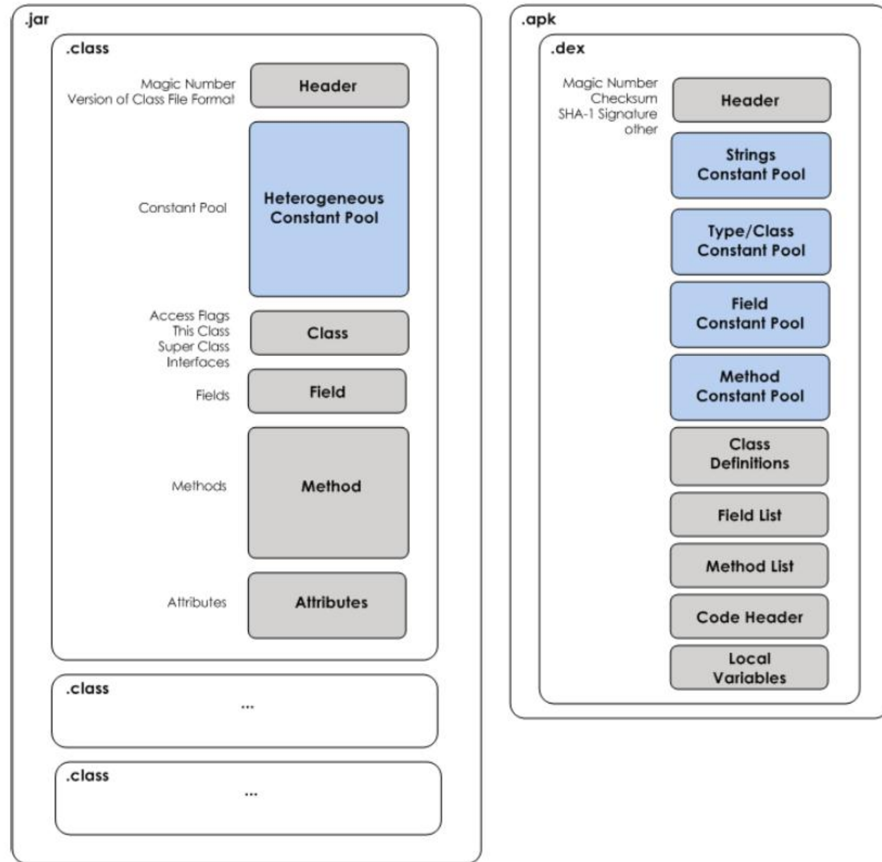


Figura 14. Diferencias entre JME y Dalvik en su estructura de archivos [Eringer 2010]

La diferencia principal radica en que la JME contiene un contenedor heterogéneo para sus constantes que representan las literales que la aplicación tiene para su ejecución, mientras que Dalvik subdivide el contenedor para tener distintos contenedores en base a los tipos de datos y solo almacena apuntadores hacia los datos [Androidconferences 2012], como se puede ver en la figura 15:

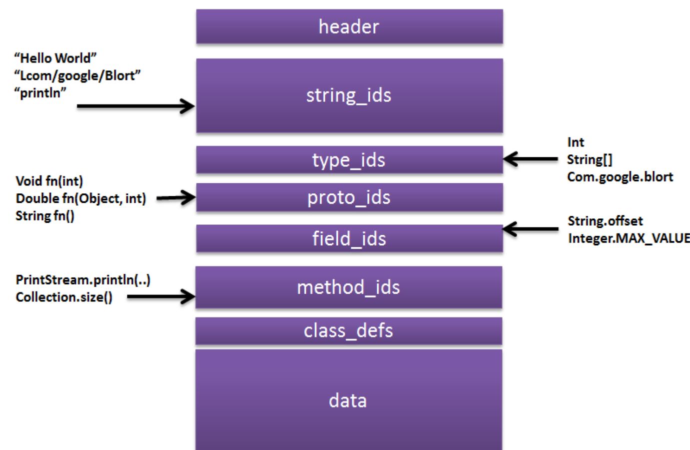


Figura 15. Anatomía de un archivo Dex

Se puede observar que los contenedores solo contienen Id's que son apuntadores hacia los datos, el archivo también puede tener definiciones de clases para indicarle que tiene otras clases con los cuales interactuar, la parte final son los datos que son mapeados por los contenedores especializados para obtener la información, la razón del por qué tener diferentes contenedores es la de hacer un manejo más rápido de la información, evitando multiplicidades y tener una compartición de información más rápida, eficiente y óptima dadas las limitaciones del Hardware [AndroidConferences 2012].

La memoria es administrada por medio de tener repetición mínima de constantes, tener contenedores especializados que implican una tipificación nata y por la indexación de los contenedores a los bloques de datos [Kumar 2008].

En la figura 16 se muestra una comparación sobre el nivel de compresión que se logra al hacer los contenedores independientes por datos y al tratarlos por medio de una indexación hacia el bloque de datos:

(U) uncompressed jar file
(J) compressed jar file
(D) uncompressed dex file

common system libraries
(U) 21445320 - 100%
(J) 10662048 - 58%
(D) 10311972 - 48%

web browser app
(U) 470312 - 100%
(J) 232065 - 49%
(D) 209248 - 44%

alarm clock app
(U) 119200 - 100%
(J) 61658 - 52%
(D) 53020 - 44%

Figura 16. Comparación del tamaño entre archivos compilados java

Claramente se observa que los archivos ".dex" aún y cuando no son comprimidos tienen una reducción tremenda respecto de los archivos de la JME, obteniendo la misma funcionalidad (ya que es el mismo código el que se ejecuta), pero obteniendo bondades al momento de ser procesados (rapidez), almacenados (distribución inteligente en disco) y consultados (ahorro de energía).

3.3.2 Tipos de memoria

En esta sección discutimos los tipos de memoria desde el punto de vista de Android. Aunque físicamente se tienen memorias basadas en tecnología NAND (las basadas en NOR son más caras y por consecuencia de poco uso) y su principal limitación para la implementación de una estructura de datos es que no soportan actualización directa: antes de actualizar un dato, primero se tiene que borrar la localidad de escritura y luego escribir el nuevo valor.

En Android se tienen cuatro tipos de memoria, a manera de tener un mejor desempeño para la vida de la batería y las condiciones del procesador [AndroidConferences 2012]:

- **Clean:** Cualquier espacio de memoria que el sistema operativo encuentra libre para utilizar.
- **Dirty:** Memoria que ya se encuentra utilizada pero se demanda que tenga otro valor.
- **Shared:** Memoria que pueden utilizar varios procesos.
- **Private:** Memoria que es exclusiva de un proceso específico.

En conclusión se tiene por un lado una arquitectura que tiene un kernel de Linux como engine primario, encargado de los drivers y de proveer un sistema de archivos y procesamiento para aplicaciones y una máquina virtual diseñada de manera óptima para hacer que las aplicaciones puedan ser ejecutadas de una manera eficiente, consciente de la restricción de la memoria, de la pobreza del procesador y de el recurso crítico de la batería.

3.3.3 Ejecución de aplicaciones en Android

Hay que considerar que Android es una plataforma que sigue el principio de seguridad de “Sandbox”, el cuál es un mecanismo para poder separar diferentes programas y que corran de manera autocontenida. Es por eso que todas las aplicaciones Android corren en su propio proceso y tienen su propia instancia de la máquina virtual Dalvik, además cada aplicación hace uso de los archivos “.dex” lo que le permite ser más eficiente y tener una mínima “footprint” (dígase el consumo de memoria que tiene una aplicación), así la máquina virtual Dalvik tiene su funcionamiento basado en hilos y permite un manejo de la memoria a bajo nivel [Ehringer 2010].

Dado que cada aplicación corre en su propia máquina virtual debe de estar lista para empezar cuando el usuario desee invocarla, por esto Android tiene un proceso que inicia cuando el sistema arranca llamado “Zygote” quien se encarga de inicializar la máquina virtual Dalvik y precarga las librerías necesarias para el funcionamiento de las aplicaciones, una vez que está inicializado trabaja en base a eventos, es decir que espera pacientemente hasta que una petición cae en algún socket, cuando llega el Zygote se encarga de levantar el proceso y las librerías necesarias para poder darle los recursos necesarios a la aplicación, además las librerías comunitarias de la arquitectura están disponibles para todas las aplicaciones con acceso solo de lectura, si alguna aplicación quisiera modificar alguna de las librerías para una funcionalidad específica se hace un “copy-on-write”[Ehringer 2010] y se le otorga una copia independiente del conjunto comunitario y se separa en su propia máquina virtual, propiciando seguridad, rapidez y agilidad en el manejo de memoria.

Si bien las plataformas móviles utilizan un sistema de bajo costo generalmente formateado en FAT (File Allocation Table) en una tarjeta de memoria [Tuch 2009], Android no es la excepción y la manera de trabajar por medio de sus archivos ".dex" es el construir un canal directo entre el procesamiento y el medio masivo de almacenamiento, a manera de dejar la memoria RAM lo más libre posible y así poder acceder por medio de apuntadores en los archivos a los datos, de manera que la memoria RAM se utiliza libremente para el almacenamiento de información necesaria de la aplicación para procesar la información en el CPU del dispositivo y volver a rescribir los archivos ".dex" o bien en alguna partición aledaña ya sea en el "itnernal storage" o bien "extarnal storage" (refiriéndose a la memoria ROM del teléfono interna o bien algún medio de almacenamiento exterior) con las actualizaciones necesarias, tomando en cuenta que una aplicación para Android está representada por el paquete global APK (*application package file*) que contiene un compendio de archivos ".dex" para su funcionalidad, la manera en que la arquitectura Android

3.4 Eficiencia en las aplicaciones Android

En esta sección se tocarán diferentes puntos a fin de obtener una aplicación eficiente que aproveche los recursos de la memoria y poder de procesamiento al máximo.

3.4.1 Optimización de código

Independientemente de la versión de Android [Guihot 2012]en la que nos encontremos o incluso la plataforma en la que nos encontremos desarrollando, la optimización de código es vital para poder impregnar una lógica clara, sin tantos ciclos y que resuelva el problema que estamos tratando de resolver, por lo tanto es de suma importancia el conocer las bondades que la plataforma nos proporciona a través de su API (Application Programming Interface).

Si bien, el optimizar el código no es una de las prioridades iniciales de una aplicación [Guihot 2012], siendo de más peso el poder lograr que la aplicación le de al usuario una buena experiencia y que a su vez se centre en la mantenimiento del código fuente. Por ende en una plataforma móvil es importante considerar:

- Funciones que contengan estructuras iterativas en vez de recursivas, ya que permiten ejecutar dentro de los ciclos más instrucciones y ser más flexibles con menos llamadas a un mismo método.
- Utilizar los tipos de datos idóneos para manejar la información que se va a procesar.
- Utilizar sistemas de cacheo para tener menos cómputo al recuperar datos.
- Para manejar caché Android provee clases como *Android.util.LruCache<K, V>*, y un tipo de arreglos destinados al cacheo llamados *SparseArray.*
- Considerar las estructuras de datos provistos por Android para manejar la información. (LruCache, SparseArray, SparseBooleanArray, SparseIntArray, Pair).
- Se debe de considerar siempre que la red y el acceso a datos son lentos.
- Si se ha de utilizar una base de datos SQLite debe de ser en base a transacciones.
- Considerar si es posible para qué dispositivos Android será la aplicación y en qué versión, en la tabla 8 se muestra un desglose de versiones con las características que se han optimizado para el performance de la plataforma Android.

API level	Version	Name	Significant performance improvements
1	1.0	Base	
2	1.1	Base 1.1	
3	1.5	Cupcake	Camera start-up time, image capture time, faster acquisition of GPS location, NDK support
4	1.6	Donut	
5	2.0	Éclair	Graphics
6	2.0.1	Éclair 0.1	
7	2.1	Éclair MR1	
8	2.2	Froyo	V8 Javascript engine (browser), JIT compiler, memory management
9	2.3.0 2.3.1 2.3.2	Gingerbread	Concurrent garbage collector, event distribution, better OpenGL drivers
10	2.3.3 2.3.4	Gingerbread MR1	
11	3.0	Honeycomb	Renderscript, animations, hardware-accelerated 2D graphics, multicore support
12	3.1	Honeycomb MR1	LruCache, partial invalidates in hardware-accelerated views, new Bitmap.setHasAlpha() API
13	3.2	Honeycomb MR2	
14	4.0	Ice Cream Sandwich	Media effects (transformation filters), hardware-accelerated 2D graphics (required)

Tabla 8 Mejoras al performance de Android [Hashimi 2010]

3.4.2 Uso de NDK (Native Development Kit)

Para escribir aplicaciones en Android se puede hacer nativamente desde C/C++, al utilizar el NDK no se tiene que pasar por la máquina virtual Dalvik y así se ejecutan directamente en el procesador, lo que hace que la aplicación sea más rápida. Además se puede combinar los lenguajes de programación, es decir se puede escribir una parte en Java y otra en C/C++ [Guihot 2012].

El NDK se agregó como característica en la versión de Android 1.5, para poder utilizar el NDK y construir totalmente las aplicaciones en C/C++ se requiere tener la versión 2.3 de la plataforma Android o superior.

3.4.3 Ciclo de vida de una aplicación Android

Para efectos de optimización en cualquier plataforma, es de suma importancia el conocer el ciclo de vida de la ejecución del software en la misma. En Android este ciclo de vida representa una máquina de estados [Herogyang 2012], la cual interactúa a manera de eventos con la plataforma, en la figura 17 ejemplifica un diagrama de estados para la plataforma Android.

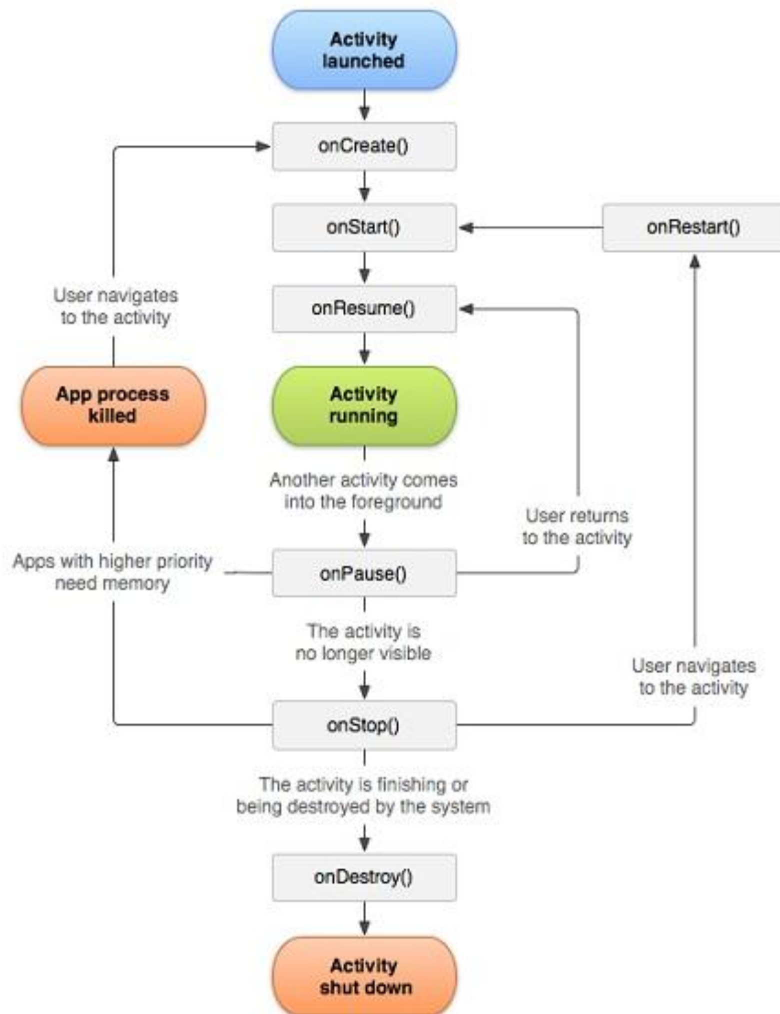


Figura 17. Ciclo de vida de una aplicación Android

Entre más rápido el ciclo de vida se complete, el usuario podrá utilizar más rápidamente su aplicación [Guihot 2012], para hacer más óptimo el ciclo de vida de cualquier aplicación es necesario:

- Reducir la complejidad en los XML, especialmente en los layout.
- Utilizar un RelativeLayout en ves de un LinearLayout anidado

- Utilizar ViewStub (es una vista que puede generar recursos en tiempo de ejecución, para cargarlos cuando son necesarios) para la creación de objetos.
- Utilizar inicializaciones flojas (se refiere a cargar solo lo necesario para que la aplicación esté disponible, y en el tiempo de ejecución se cargará lo restante siempre y cuando se ocupe).

3.4.4 Utilizando la memoria eficientemente

Las aplicaciones en Android gastan una parte muy significativa de su tiempo tratando con datos en la memoria [Guihot 2012], por eso es importante el realizar una aplicación que sea consciente de los recursos del dispositivo y tenga consideraciones bajas en la utilización de la memoria sin perjudicar su desempeño y comodidad para el usuario.

Es importante tener en cuenta las siguientes consideraciones:

- No importa la cantidad que se le asigne a una aplicación Android, ésta siempre pedirá más.
- Android no es capaz de manejar memoria SWAP (o de compartición con el disco duro).
- Tener en cuenta que si la aplicación se desarrolla en java, tendremos el siguiente espacio utilizado por nuestras variables y constantes (Tabla 9):

Tipo primitivo de Java	Tipo nativo	Tamaño
boolean	jboolean	8 bits (VM dependiente)
byte	Jbyte	8 bits
char	jchar	16 bits
short	jshort	16 bits
int	jint	32 bits
long	jlong	64 bits
float	jfloat	32 bits
double	jdouble	64 bits

Tabla 9 Tipos primitivos en Android

- Utilizando sabiamente la memoria se puede incrementar el “performance” de la aplicación, para ello podemos utilizar los tipos de datos primitivos de la tabla 3 para que satisfagan nuestra aplicación.
- El “performance” es derivado principalmente de tres factores: El primero de cómo el CPU manipula ciertos tipos de datos; el segundo del espacio necesario para almacenar datos e instrucciones; y el tercero de cómo los datos son presentados en la memoria.
- El trabajar con tipos de datos que ocupan 64 bits es más lento que trabajar con tipos de datos de 32 bits [Guihot 2012].

- Para manejar arreglos es mejor trabajar con datos de 16 bits, ya que tienen algoritmos predefinidos acorde a su tamaño como el ordenar, sortear, entre otros, a fin de hacerlo más efectivo.
- Evitar el “Casteo” (convertir un tipo de dato en otro).
- Android cuenta con dos niveles de caché, L1 y L2: el primero es el más rápido, pero a su vez el más pequeño, al contar con tan solo 64 kilobytes (32 kb para el caché de datos y 32 kb para el caché de instrucciones), mientras que el segundo caché cuenta con 512 kb [Guihot 2012].
- Es importante considerar que para el manejo de caché también existen pérdidas, de instrucciones, de datos o bien de escritura, lo que reduce notablemente el performance de la aplicación al tener que ir a recuperar la información hasta que sea leída de la memoria.
- Utilizar el menor encapsulamiento posible.

Existe una perspectiva en la plataforma Eclipse llamada DDMS, la cual nos indica el uso de memoria de nuestras aplicaciones. Existe una API para el Garbage Collector que la máquina Dalvik [Egringer 2010] maneja, podemos invocarlo con la instrucción “System.gc”, pero la máquina Dalvik se reserva el derecho de ejecución para la instrucción, cuando éste así lo decida. Existen 5 situaciones en las que el colector de basura entra en acción:

1. GC_FOR_MALLOC: Ocurre cuando la pila está muy llena para acomodar memoria y la memoria debe ser reclamada antes de que el acomodo se pueda llevar acabo.
2. GC_CONCURRENT: Ocurre cuando una colección de objetos es demasiado grande y ya no se utiliza.
3. GC_EXPLICIT: Ocurre cuando se manda llamar directamente al colector “System.gc”.
4. GC_EXTERNAL_ALLOC: No ocurre desde la versión honeycomb
5. GC_HPROF_DUMP_HEAP: Ocurre cuando se crea un archivo de tipo HPROF (archivos de diagnóstico de la pila de memoria [9]).

Android, además cuenta con una API para acceder a los valores de la memoria, tales como:

- ActivityManager’s getMemoryInfo()
- ActivityManager’s getMemoryClass()
- ActivityManager’s getLargeMemoryClass()
- Debug’s dumpHprofData()
- Debug’s getMemoryInfo()
- Debug’s getNativeHeapAllocatedSize()
- Debug’s getNativeHeapSize()

3.4.5 Overhead

El overhead en Android es muy alto, ya que requiere levantar una máquina virtual Dalvik aun y cuando no se utilice por completo toda la suite de funciones que se tenga, tomando en cuenta que la plataforma es limitada y es lenta se aconseja a manejar lenguajes nativos como C/C++ que

permiten reducir el overhead que es variable entre cada una de las aplicaciones, pero a la vez no es recomendable debido a que la seguridad se vería afectada.

4. Capítulo Cuarto: Métodos de Persistencia

La persistencia es el mecanismo que permite leer y escribir información en un medio físico. La persistencia es vital para las estructuras de datos que manejan información, por lo tanto, el B+-Tree debe de contar con un mecanismo para hacer persistente los datos, a manera que estén disponibles para su consulta, eliminación o nuevas inserciones [Brodal 2012].

La motivación es la de encontrar la manera idónea de realizar la persistencia para un algoritmo B+-Tree de manera que consuma pocos recursos del dispositivo y sea eficiente.

La contribución del capítulo es el de encontrar de manera empírica el mejor método para implementar la persistencia para un B+-Tree en una plataforma móvil Android, analizando mecanismos que permitan implementar la persistencia y seleccionando el más óptimo.

4.1 Análisis de implementación de Persistencia

La persistencia es la acción de preservar y poder leer la información de un objeto de forma permanente, en la implementación de cualquier aplicación para plataformas móviles la persistencia juega un rol muy importante para almacenar datos esenciales del usuario y así poder manipular la información almacenada [Lanka 2011]. En éste análisis se investigaron tres métodos para implementar persistencia, los cuales se describen brevemente a continuación, posteriormente se contrastan para seleccionar el mejor para ser parte de un B+-Tree en una aplicación para la plataforma móvil Android:

4.2 API de persistencia en Java

Java cuenta con una API de persistencia que simplifica el modelo de persistencia, la API negocia con datos mapeados hacia objetos Java llamándolos “Entidades persistentes” (persistent entities) [OracleEntity 2012], implementando un modo para poder acceder a los objetos almacenados en bases de datos relacionales para que puedan ser accedidos en un futuro, además de estandarizar el mapeo relacional de objetos de Java.

Es compatible con EJB (Enterprise Java Beans) y puede funcionar dentro o fuera de él. La ventaja de éste método sería la creación de las entidades necesarias por medio de un Factory (fábrica de entidades para la base de datos manipulada desde el código fuente) para hacer que el B+-Tree funcione en una base de datos, sin embargo el API no es idóneo debido a que crea su propia estructura de archivos a manera de base de datos y la funcionalidad del algoritmo B+-Tree quedaría completamente inservible.

4.3 Serialización de Objetos

Java posee los módulos necesarios para poder almacenar y retornar objetos Java por medio de la “serialización”, ésta puede ser orientada a flujos de red o bien a archivos, consiste simplemente en

encapsular un objeto en bytes para así poder manejarlo en un buffer y decidir qué hacer con él, una vez que el objeto es reconstruido [OracleSerialization 2012].

La serialización es bastante rápida en tipos de datos primitivos, aunque en objetos es un tanto variable debido a las relaciones de herencia [Lee], agregación o composición que pueda tener un objeto respecto de otros, estos factores se ven afectados en el tiempo de serialización y el tamaño del bloque de bytes que despida.

4.4 Persistencia a bajo nivel

Utilizando Java se puede escribir archivos básicos que permitan expandir las posibilidades del B+-Tree de manera amplia, no cargando demasiadas librerías y siendo un poco más libre en su ejecución, consulta, mantenimiento y funcionalidad.

Con el uso de buffers y manejo de archivos se puede realizar la persistencia en archivos binarios que permita al B+-Tree en una plataforma móvil el ser un poco más óptimo.

4.5 Contraste de métodos para implementar Persistencia

El análisis refleja la decisión de elegir un método para implementar la persistencia para el B+-Tree que se desarrolló para la plataforma Android, el cuál necesita una estructura que tenga acceso aleatorio al disco, el método de persistencia debe de ser cuidadoso con la memoria RAM: debido a que consumir recursos de memoria y procesamiento son importantes para otras aplicaciones y para proveer un aplicación amigable para el dispositivo móvil. Las tablas siguientes contienen un análisis FODA respecto a los métodos de persistencia candidatos a ser el óptimo.

Método Característica	API Persistencia JAVA	Persistencia de Objetos (Serialización)	Persistencia a bajo nivel
Fácil acceso a la información		✓	✓
Fácil implementación		✓	
Digestión en memoria simple		✓	✓
Velocidad de procesamiento		✓	✓
Consideraciones energéticas		✓	✓
Complejidad en el uso	✓		

Tabla 10. Comparación de métodos para implementar la persistencia

	Fortalezas	Debilidades
Dentro de Android	• La implementación sería muy rápida. • Las librerías de Java ya están pre-programadas. • El B+-Tree podría emular el comportamiento para insertar en la base de datos.	• El algoritmo B+-Tree dejaría de ser el mismo al convertirse en un híbrido que cohabite con una base de datos manipulada por Java. • Se tendría que programar un módulo de querys que permitan manipular la

		información.
Fuera de Android	• La implementación podría ser exportable a cualquier plataforma que utilice Java.	• Viola los principios de un B+-Tree al implementar otro algoritmo de indexación y acceso a datos.

Tabla 11. FODA API de persistencia Java

	Fortalezas	Debilidades
Dentro de Android	• Implementación sencilla • Acceso a la información inmediata una vez que el objeto esté levantado. • Los objetos conocen su meta-información por lo que el almacenado en disco es inmediato.	• Se tendría que hacer varias pruebas para poder implementarlo ya sea a nivel de los nodos, a nivel de índice o el árbol en su totalidad.
	Oportunidades	**Amenazas**
Fuera de Android	• La implementación podría llevarse a cabo para un B+-Tree en memoria para una plataforma que no sea móvil.	• No se tienen amenazas.

Tabla 12. FODA Persistencia de Objetos (serialización)

	Fortalezas	Debilidades
Dentro de Android	• No utiliza librerías ajenas, por lo que solo se concentraría en cargar lo necesario. (ahorro de vida batería, agilidad de procesamiento, consumo de memoria).	• Implementación laboriosa debido al uso de buffers y diversidad de archivos para indexar los datos.
	Oportunidades	**Amenazas**
Fuera de Android	• La implementación podría ser exportable a cualquier plataforma que utilice Java.	• Que la memoria colapse si no se limpian cuidadosamente los buffers. • Que el dispositivo se sobrecargue y se reinicie, por la acumulación de basura en la memoria, esto va de la mano con la capacidad de cada dispositivo. • Que el dispositivo funcione lentamente.

Tabla 13. FODA Persistencia a bajo nivel

Por lo tanto consideramos a la serialización el mecanismo más factible para realizar la persistencia, ya que es rápido en cuanto a su procesamiento, lo que implica gastos mínimos para el procesador/memoria y fácil de implementar. Además la serialización es una herramienta que en un futuro permitiría múltiples ventajas para el árbol, por ejemplo, el transmitirlo por red.

5. Capitulo Quinto: Diseño e implementación de un B+-Tree

En este capítulo se encuentran detalles del diseño y de la implementación que se realizaron para tres instancias de un B+-Tree, cada una de ellas cumple con las operaciones básicas de inserción, eliminación, consulta y modificación, la diferencia principal de las instancias radica en su método de persistencia: la primera utiliza serialización para los nodos; la segunda utiliza serialización en toda la estructura del árbol; y la tercera implementa el uso de archivos para mantener los datos, utilizando archivos de acceso aleatorio.

La motivación de las diversas implementaciones, surge para comparar la mejor manera en la que un B+-Tree puede implementarse dentro de una plataforma móvil, utilizando la plataforma Android.

El capítulo está dividido en tres partes: la primera es el desarrollo de las implementaciones que utilizan serialización, tanto en los nodos hoja, como en el árbol, respectivamente; la segunda muestra el desarrollo de la implementación que utiliza archivos; la tercera refleja la utilización del B+-Tree a través de una interfaz gráfica que es compartida por todas las implementaciones.

5.1 Implementaciones basadas en Serialización

5.1.1 Diagrama de componentes

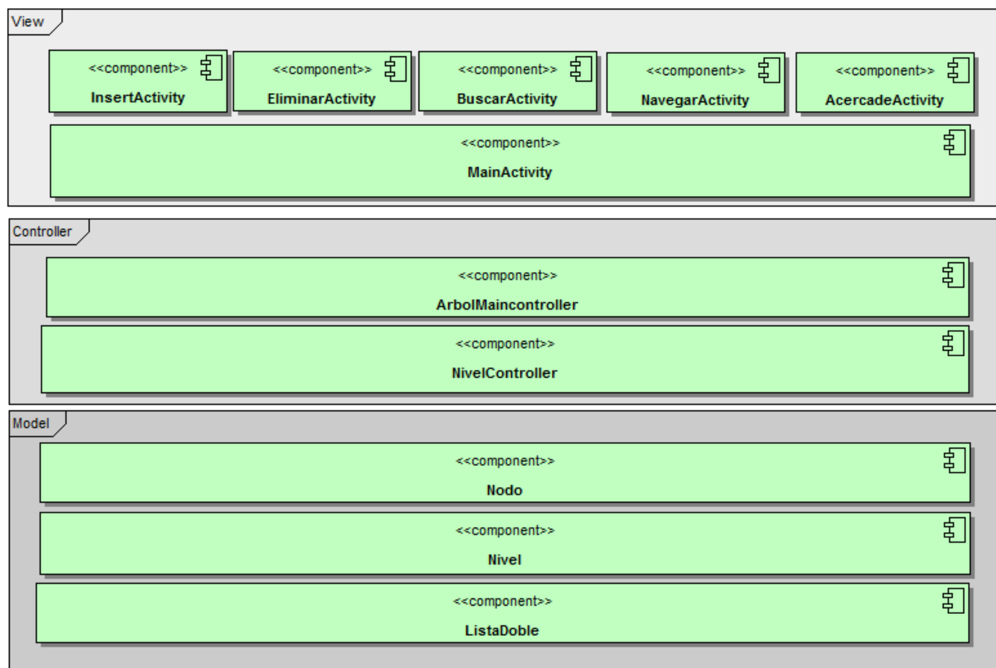


Figura 18. Diagrama de Componentes B+-Tree Android

La figura 18 denota el diagrama de Componentes, los cuales están soportados por las implementaciones que utilizan serialización, la tabla 14 contiene la descripción de cada uno de los componentes:

Componente	Descripción
Nodo	Componente de más bajo nivel en donde se Guardará la información de la llave y por el momento en los datos se guardará una String, en un futuro se piensa poner cualquier objeto de datos (llámese estructura, tabla, archivo, etc). Se deja en String la parte de datos para que cumpla con el requisito Alfanumérico del B+-Tree
ListaDoble	Es la lista que se forma en el último nivel del B+-Tree, la cual apela para las búsquedas que eficienta éste algoritmo de manera que no solo busca secuencialmente sino que soporta una búsqueda por medio de Rangos basado en las llaves de los nodos.
Nivel	Representa cada hoja del B+-Tree, fungiendo como un contenedor de nodos ya sean índices u hojas; por ejemplo el nodo Root puede tener 3 objetos de información que a su vez son nodos del mismo.
NivelController	Es el controlador de los niveles, maneja las inserciones, las eliminaciones y provee el manejo de los nodos a través de su nivel.
ArbolMaincontroller	Representa la interfaz que instancia al nivelcontroller y provee un acceso entendible para la vista.
MainActivity	Es la actividad principal con interfaz Android que establece una Intención (Intent) que muestra las diferentes Opciones (Eliminar, Insertar y Buscar) para que se comunique con el ArbolMainController.
InsertActivity	Soportada por MainActivity nos permite una interfaz para insertar en el árbol.
EliminarActivity	Soportada por MainActivity nos permite una interfaz para eliminar en el árbol.
BuscarActivity	Soportada por MainActivity nos permite una interfaz para buscar en el árbol por llaves específicas o por un rango de llaves.
NavegarActivity	Soportada por MainActivity permite navegar a través del árbol.
AcercadeActivity	Información general de la aplicación.

Tabla 14. Descripción de componentes

5.1.2 Diagrama de Clases

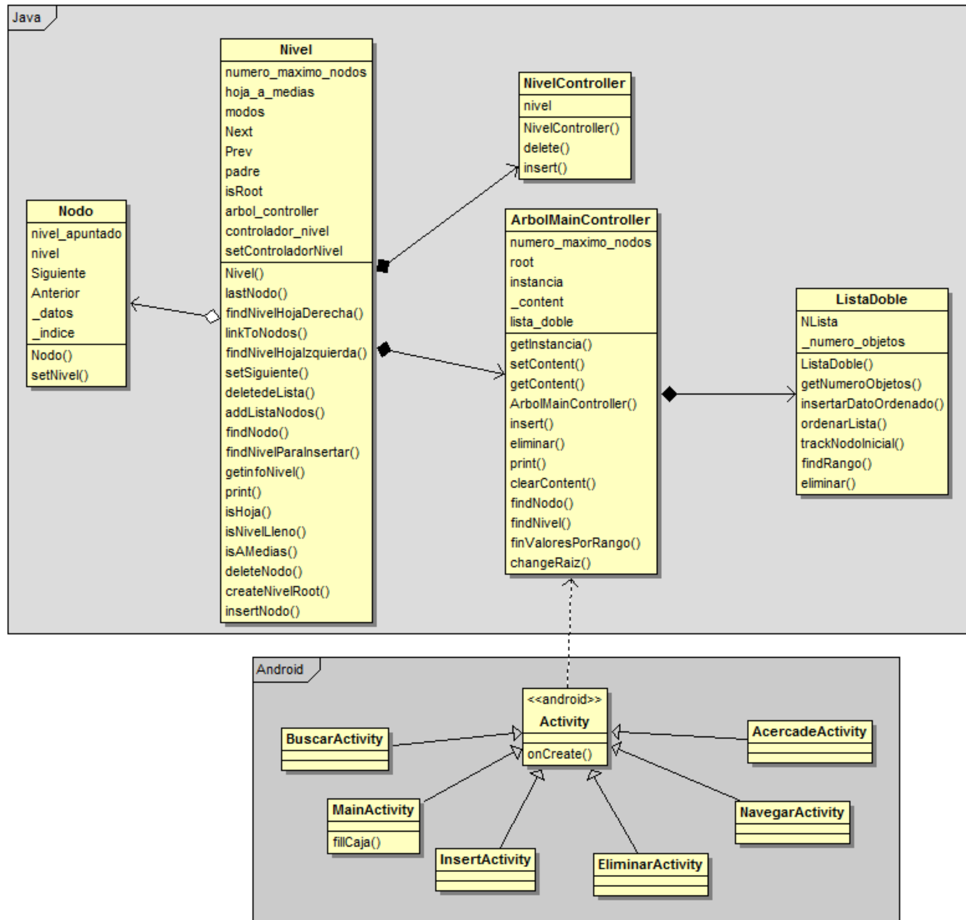


Figura 19. Diagrama de Clases B+-Tree Android para implementaciones por serialización

La figura 19 denota el diagrama de clases base que servirá para conformar la estructura de las implementaciones que utilizarán serialización. A continuación se describe cada una des las clases a detalle.

5.1.3 Clase Nodo

Componente	Descripción
Atributo::nivel_apuntado	Nivel al cual el nodo pertenece... en la documentación me refiero como un Objeto de información [ROOT] N1 N2 [HOJA_INDICE_N1] [HOJA_INDICE_N2] --> Cada hoja es un nivel apuntado por un Nodo N1 N2 N3 N1 N2 N3
Atributo::Nivel	Es el nivel al que pertenece el nodo [ROOT] --> Nivel N1
Atributo::Siguiente	Puntero para efectos de ListaDoble y propósitos de Búsqueda por rango

Atributo::Anterior	Puntero para efectos de ListaDoble y propósitos de Búsqueda por rango
Atributo::_datos	Para respetar el algoritmo B+-Tree se dejó como Alfanumérico aunque aquí podemos crear una interfaz para cualquier tipo de objeto
Atributo::_indice	Representa la key, llave o índice por el cual vamos a indexar
Método-->Nodo (in indice : int)	Se inicializan los datos con la llave que creará al nodo y se inicializa datos en una cadena vacía

Tabla 15. Descripción clase Nodo

5.1.4 Clase Nivel

Componente	Descripción
Atributo:: numero_maximo_nodos	El número máximo permitido de nodos (Objetos de información) en el nivel. [ROOT] --> Nivel N1 N2 N3 numero_maximo_permitido = 3
Atributo:: hoja_a_medias	Apela a la regla de 1/2 del B+tre por nivel. La cual dice que cada Nivel debe de contener por lo menos el 0.5 de su capacidad para vivir en el árbol
Atributo::nodos	La colección de nodos que contiene el nivel [HOJA_INDICE] [N1, N2, N3 ... Ni] --> Colección de nodos
Atributo::Next	Puntero para recorrer los niveles como una lista cuando se navega por el árbol
Atributo::Prev	Puntero para recorrer los niveles como una lista cuando se navega por el árbol
Atributo::padre	Nos dice quien es el padre de éste nivel
Atributo::isRoot	Nos dice si el nivel es la raíz de nuestro B+-Tree
Atributo::árbol_controller	Nuestro controlador de niveles inmediato
Método--> Nivel(in numero_maximo_nodos : int)	Setea el número máximo de nodos para éste nivel. Inicializamos nuestra colección de nodos Inicializamos el valor de la hoja media para los nodos apelando al 0.5 mínimo
Método--> lastNodo()	Regresamos el último nodo de nuestro nivel [ROOT] n1 n2 n3 --> n3
Método--> findNivelHojaDerecha()	Regresamos el nivel al que apunta el último nodo de la hoja derecha, esto es muy útil para el reacomodo de los punteros cuando existe un copy-up y un push-up
Método--> findNivelHojaIzquierda()	Regresamos el nivel al que apunta el último nodo de la hoja izquierda esto es muy útil para el reacomodo de los punteros cuando existe un copy-up y un push-up
Método--> linkToNodos(in ultimo : Nivel)	Migramos el apuntador del nodo al que se moverá

Método--> setSiguiente(in nivel : Nivel)	Nos movemos al nivel siguiente, útil para cuando queremos hacer el debug de nuestro árbol
Método--> deletedeLista(in nodo : Nodo)	Quitamos el nodo del nivel de nuestro Vector nodos
Método--> addListaNodos(in nodo : Nodo)	Agregamos el nodo a nuestra colección de tipo Vector en nuestro nivel
Método--> findNodo() : Nodo	Regresamos el Nodo encontrado en nuestra colección nodos del nivel si existe
Método--> findNivelParaInsertar(in indice : int) : Nivel	Recorremos los niveles para ver cuál es el nivel hoja y regresarlo para que se realice la inserción del nuevo nodo, Representa el Search del B+-Tree para llegar al nodo hoja Parametrizada con el nodo de búsqueda (Nivel) [recursivo] y con la llave(s) a buscar ya que puede manejar el rango de llaves para regresar múltiples valores. Se hace una búsqueda lógica comparando la llave desde la raíz hasta el último nivel que son las hojas (Si la altura del árbol es 1 la raíz es el nivel de hoja) Se obtiene la primer llave (si existe) y se regresa el valor Si la búsqueda pide un rango, se consultan los valores haciendo una búsqueda secuencial en la lista enlazada.
Método-->getinfoNivel() : string	Regresa la información del nivel para ser presentado en interfaces te texto
Método--> print(in tabuladora : int) :	Nos llena la instancia del controlador principal para poder presentarlo a las interfaces pertinentes
Método--> isHoja() : bool	Nos dice si el nivel es hoja
Método--> isNivelLleno() :bool	Nos dice si el nivel está lleno, es útil para ver cuándo vamos a hacer reacomodos mediante copy-up y push-up
Método-->isAMedias() :bool	Nos dice si la hoja esta a medias acorde a la regla del B+-Tree de 0.5 full
Método-->deleteNodo() :void	Lo votamos de nuestro nivel y reestructuramos en el método del controlador de nivel delete
Método-->createNivelRoot(in nivel : Nivel) :	Cuando se hace una restructuración y se cambia la raíz se crea un nivel root para la nueva raíz
Método-->insertNodo() :	Utilizamos el controlador de niveles para restructurar si es necesario e insertar el nuevo nodo
Método-->setControladorNivel : ArbolMainController:void	Se setea el controlador primario que tendrá contacto con la raíz

Tabla 16. Descripción clase Nivel

5.1.5 Clase NivelController

Componente	Descripción
Atributo::Nivel	Nivel a controlar por instancia
Método--> NivelController(in nivel : Nivel)	Constructor en el que setea el nivel a controlar
Método--> delete(in nodo : Nodo)	Se hace una búsqueda de la llave que nos regrese el nodo hoja donde debemos de eliminar el objeto de información.

	Eliminamos el objeto si es que se encuentra Comprobamos la altura del árbol si es 1 entonces todo es correcto porque lo quitó del nodo raíz. Si la altura no es 1, entonces mientras el nodo retrocede en el árbol haciendo un underflow se reacomodan los routers y se hacen push ups para reacomodar el árbol y dejarlo balanceado solo si un nodo tiene más entradas que las mínimas ocupadas. En caso de que el nodo no tenga las entradas mínimas se mezclan los objetos de información del nodo con el contiguo y se decrementa la altura del árbol.
Método--> insert(in nodo : Nodo)	Le mandamos el nodo a insertar Se hace una búsqueda de la llave que nos regrese el nodo hoja donde debemos de insertar el objeto de información. Se inserta el objeto de información. Si el nodo donde se insertó tiene overflow entonces se hace el proceso de copy-up y de push-up Si el nivel es 1 se crea un nuevo nodo índice como root y se separa en dos nodos índices.

Tabla 17 Descripción clase NivelController

5.1.6 Clase ArbolMainController

Componente	Descripción
Atributo::numero_maximo_nodos	El número máximo permitido de nodos (Objetos de información) en el nivel: [ROOT] --> Nivel, N1 N2 N3 numero_maximo_permitido = 3
Atributo::root	El nivel en donde descansa la raíz del árbol
Atributo::instancia	Representa el singleton del B+-Tree
Atributo::_content	Representa el contenido del árbol en formato String para proveerlo a las interfaces visuales
Atributo::lista_doble	Variable que contiene la lista doblemente ligada para poder hacer la búsqueda por rango
Método--> getInstancia() : ArbolMainController	Regresa una instancia aplicando el patrón de diseño Singleton para tener un solo árbol B+-Tree en memoria
Método--> setContent(in content : string) :void	Se va guardando la estructura del árbol en formato string para proveerlo a las interfaces
Método--> getContent() : string	Regresa la estructura del árbol en modo texto para las interfaces
ArbolMainController(in numero_maximo_nodos : int) :ArbolMAinController	Constructor Inicializamos nuestra variable máxima. Inicializamos nuestra lista Seteamos el nivel de root
insert(in indice : int, in valor : string) :void	Método alias para insertar que se comunica con el nivel adecuado y la lista para albergar el nuevo objeto de información
eliminar(in indice : int) :void	Método alias para eliminar comunicandose con el nivel adecuado y eliminando el nodo si este existe
Metodo-> print() :String	Recorre el arbol nivel por nivel y obtiene un string que representa la estructura del árbol que será mostrada a las interfaces

Metodo->clearContent() :void	Eliminamos el contenido string del árbol para generar uno Nuevo
Metodo-> findNodo(in indice : int) : Nodo	Método alias para encontrar cualquier nodo en el árbol si existe
Metodo->findNodo(in indice : int) : Nodo	Método alias para encontrar cualquier nodo en el árbol si existe
Metodo->findNivel(in indice : int) :Nivel	Método alias para encontrar el nivel al que corresponde el siguiente nodo
Metodo->finValoresPorRango(in inferior : int, in superior : int) :String	Busca en la lista enlazada el rango que se desea
Metodo->changeRaiz(in nivel : Nivel) :void	Cambiamos a que otro nivel sea la raíz del árbol

Tabla 18. Descripción clase ArbolMainController

5.1.7 Clase ListaDoble

Componente	Descripción
Atributo::Lista	Para el inicio de nuestra lista
Atributo::_numero_objetos	Para debug
Método->ListaDoble:ListaDoble	Constructor donde inicializamos Lista = null
Método-> getNumeroObjetos() :int	Efectos de Debug
Método-> insertarDatoOrdenado(in nodo : Nodo) :void	Insertamos un nodo nuevo y reacomodamos la lista
Método->ordenarLista() :void	Ordenamos la lista enlazada doble por la derecha
Método-> trackNodoInicial() :void	Recorremos hasta el primer nodo iendonos por la izquierda
Método->findRango(in inicial : int, in final : int) : string	Encontramos el rango pedido y retornamos los valores a manera String para que las interfaces lo utilicen
Método->eliminar() :void	Eliminamos un nodo de la lista

Tabla 19. Descripción clase ListaDoble

5.1.8 Clase Activity

Clase propietaria de Android de la cual se cuelga el B+-Tree para poder hacer intenciones y así poder dar flujo a la aplicación, el método primordial de ésta clase es el onCreate() que es cuando se levanta la actividad por medio de una intención, las demás clases heredan de ésta y apelan a lo mostrado en el capítulo de Casos de uso sobrescribiendo el método onCreate() para los propósitos que cada una de ellas ocupe, las interfaces son dictadas por archivos XML que contienen las cajas de texto, las variables de texto y demás.

5.1.9 Actividades

Los diagramas de actividades denotan la interacción de los objetos instanciados de las clases antes descritas, por ejemplo, se puede apreciar que en la figura 20, se hace una inserción, que atraviesa desde la plataforma Android pasando por su controlador y terminando en su modelo en memoria, la arquitectura MVC nos permite hacer la separación para guardar y mostrar resultados en las interfaces Android.

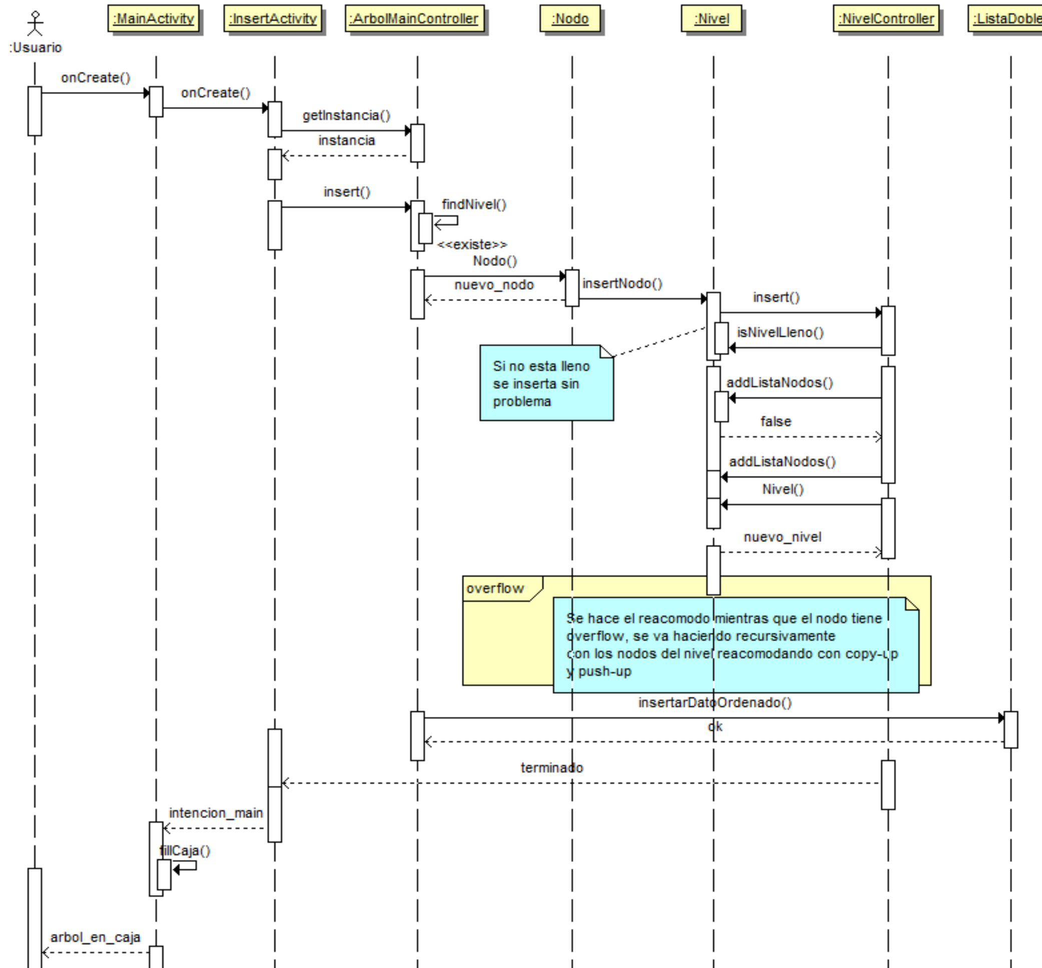


Figura 20. Insertando en un B+-Tree

La búsqueda sencilla puede apelar a buscar tanto un índice como un valor, en la figura 21 se puede apreciar la interacción entre los objetos.

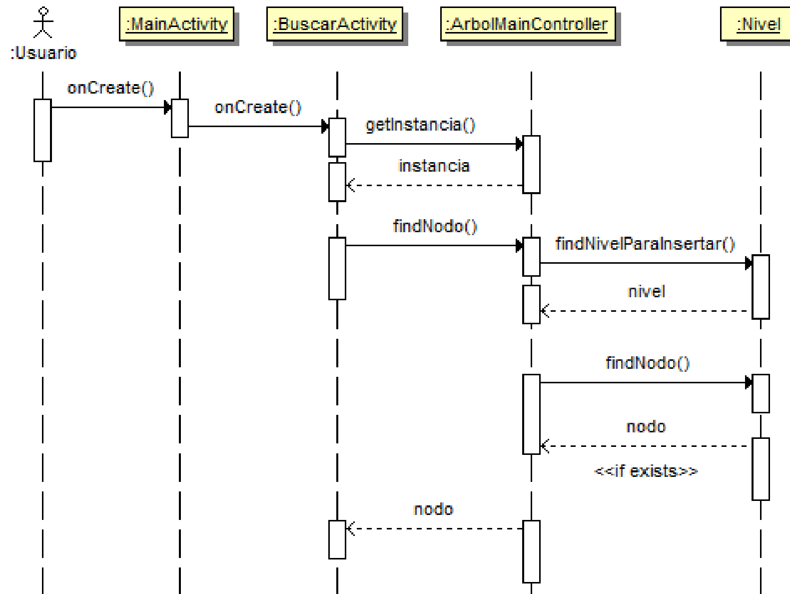


Figura 21. Búsqueda sencilla

La búsqueda por rango es una de las ventajas inminentes y una de las características principales del B+-Tree, en la figura 22 podemos ver, el cómo se navega entre objetos instanciados a manera de buscar en la lista enlazada doble los resultados comprendidos entre un rango definido de índices.

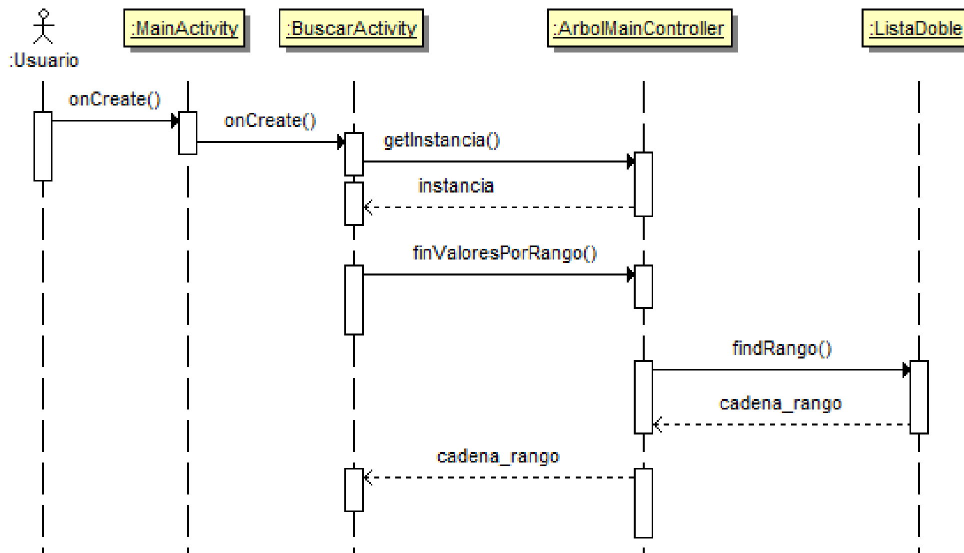


Figura 22. Búsqueda por rango

Tanto la inserción como la remoción de nodos del árbol se crea un constante reacomodo por medio de los algoritmos copy-up y/o push-up según sea el caso, mismos que ya se han analizado, en la figura 23 mostramos el diagrama de eliminación del árbol B+-tree apelando al flujo lógico de nuestros objetos.

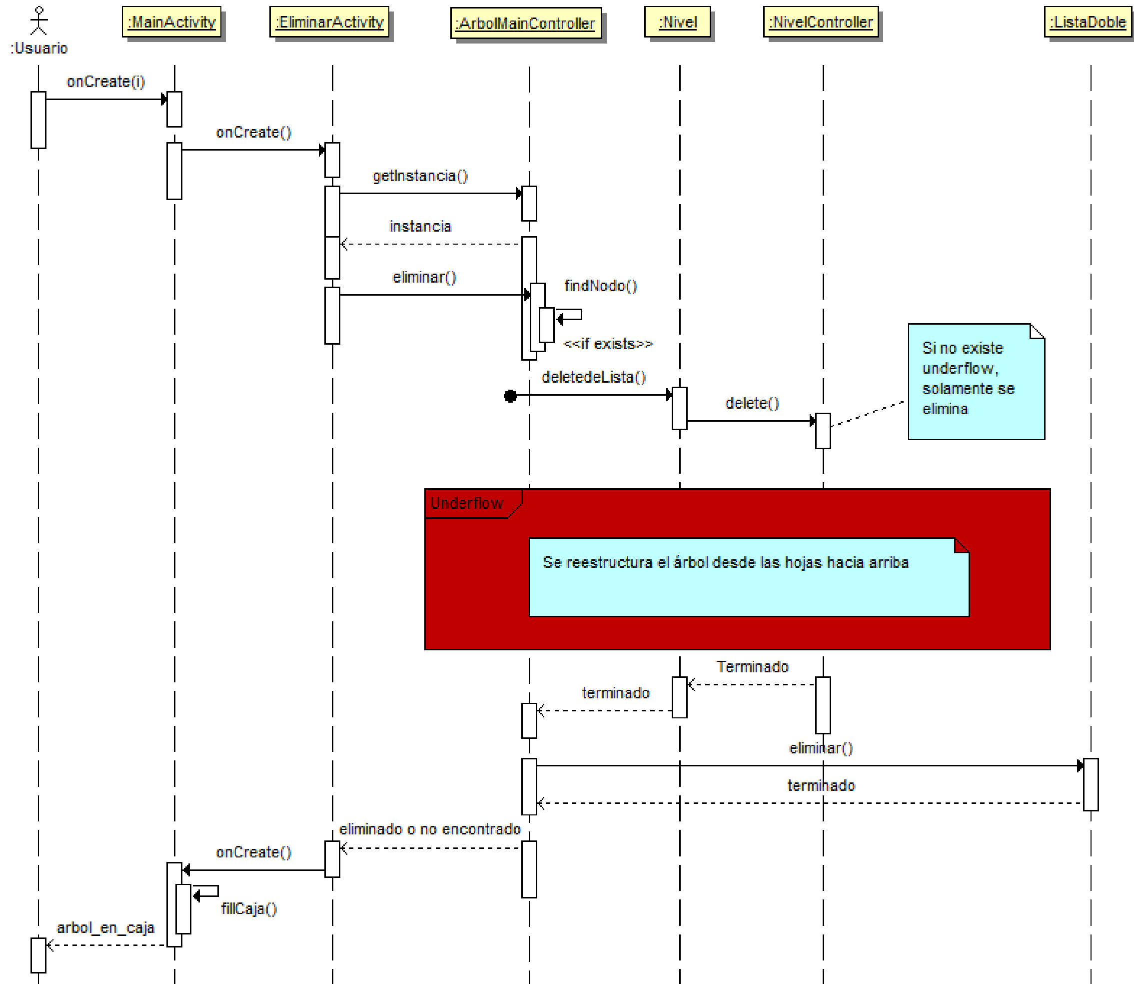


Figura 23. Eliminación de un nodo B+-Tree

5.1.10 Persistencia

Una vez descrito el algoritmo base, se procede a implementar los métodos de persistencia, el diseño de la persistencia se bifurca en dos modalidades, la primera apela al manejo de los índices en un archivo y de los nodos serializados como lo muestra la figura 24:

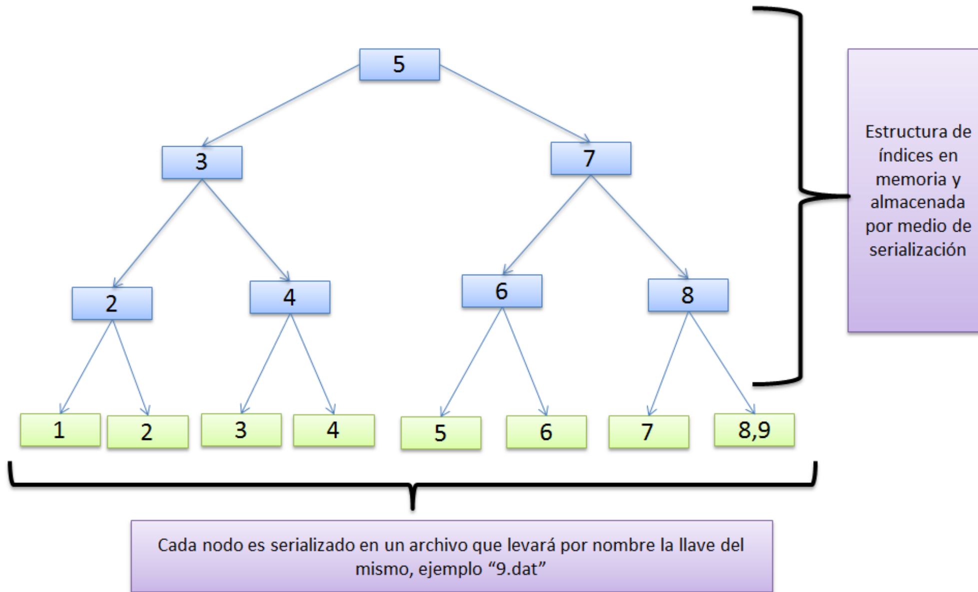


Figura 24. Implementación con nodo serializado

Mientras que la segunda es la serialización total del árbol B+-Tree, como se muestra en la figura 25:

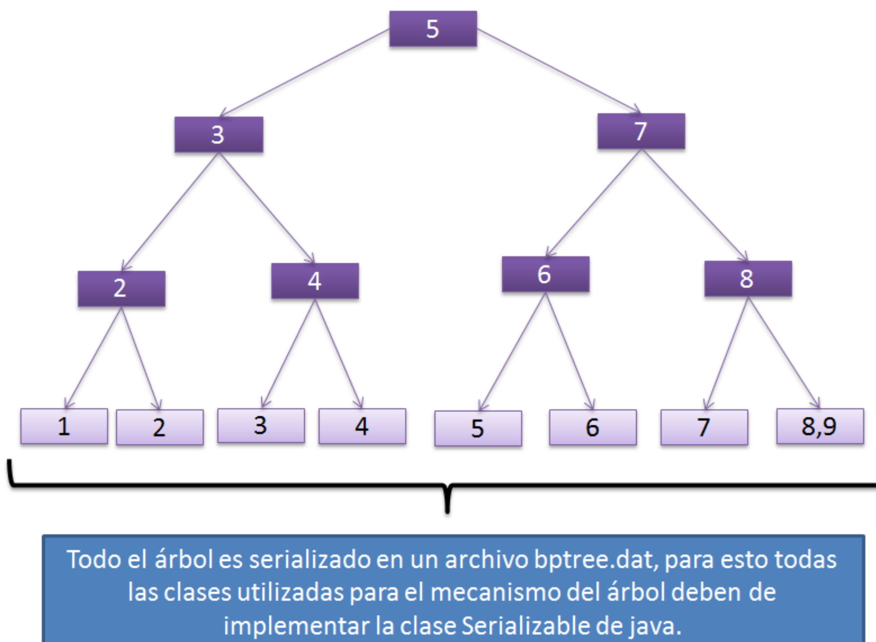


Figura 25. Serialización de todo el B+-Tree

5.1.11 Implementación

Se implementa en base a código escrito en el lenguaje java una aplicación para un dispositivo móvil estándar que realice el funcionamiento de un B+-Tree. La motivación es proporcionar una estructura eficiente que permita la indexación en dispositivos móviles, teniendo como objetivo el implementar las operaciones básicas; inserción, eliminación y búsqueda, atendiendo a las limitaciones de la plataforma móvil, a fin de considerarlas y aprovechar los recursos habidos al máximo.

Las contribuciones que obtendremos del capítulo será el desarrollo de la implementación de un B+-Tree para la plataforma Android. La tabla 20 muestra las etapas de implementación que fueron necesarias para la realización del prototipo:

Parte	Descripción
Implementando el algoritmo de búsqueda	Abordará el cómo implementar el algoritmo de búsqueda y la lista enlazada que hace potente al B+-Tree
Implementando el algoritmo de inserción	Relata el cómo se altera el árbol B+-Tree por medio de inserciones.
Implementando el algoritmo de eliminación	Trata del cómo se altera el árbol B+-Tree por medio de eliminaciones.
Implementando la persistencia	Comprende las diferentes maneras de implementar la persistencia acorde al diseño.

Tabla 20 Layout del análisis de algoritmos

5.1.12 Implementando el algoritmo de búsqueda

Teniendo conocimiento de cómo se manejan las intenciones y actividades en Android, se limitará a la funcionalidad específica descrita en las clases que hacen funcionar al B+-Tree.

Se analiza la función de búsqueda para hacer match con el algoritmo descrito en el capítulo primero, Tabla 6 Algoritmos *para un B+-Tree*, acompañado por su pseudocódigo en la sección de Algoritmos en el capítulo de búsqueda; la implementación de la búsqueda se separó en dos partes, la primera se utiliza para encontrar el nivel en el que interactuaremos, es decir dada una llave *x* tenemos que llegar hasta su nivel hoja, por lo que una primera función nos devuelve la posición en la que se encuentra ese nodo (o bien el nivel), y se hace un método alias para controlarlo desde la instancia principal en el archivo *ArbolMainController.java*, como se muestra en la figura 26.

```
103
104    /**
105     * Método alias para encontrar el nivel al que corresponde el siguiente nodo
106     * @param indice
107     * @return
108     */
109    Nivel findNivel(int indice){
110        return root.findNivelParaInsertar(indice);
111    }
```

Figura 26. Alias para encontrar el nivel a insertar (ArbolMainController.java)

El alias mapea a la función que se muestra en la figura 27, que nos devuelve el nivel en el que se encuentra el nodo que deseamos insertar, eliminar o buscar, cabe hacer mención que esta función está optimizada para plataformas móviles, al deslindarla de la recursión.

```
135    /**
136     * Encontramos donde podemos insertar
137     * al nuevo nodo regresando el nivel de hoja
138     **/
139    Nivel findNivelParaInsertar(int indice){
140        Nivel n = this;
141        while ( n.isHoja() == false   ){
142            boolean exito = false;
143            for (int i=0;i < n.nodos.size() ;i++ ){
144                Nodo nodo_temporal = n.nodos.elementAt(i);
145                if (nodo_temporal._indice > indice ){
146                    Nodo nodo_temporal2 = nodos.elementAt(i-1);
147                    n = nodo_temporal2.nivel_apuntado;
148                    exito = true;
149                }
150            }
151            if (exito == false){
152                Nivel nivel=((n.nodos.lastElement())).nivel_apuntado;
153                n = nivel;
154            }
155        }
156        return n;
157    }
```

Figura 27. Función encontrar Nivel (Nivel.java)

En la figura 28 se muestra el código para buscar un nivel en el archivo Nivel.java que apela a parte del algoritmo descrito en el Capítulo primero sección Algoritmos en el capítulo de búsqueda, los parámetros son diferentes ya que la implementación se hizo de otra manera, pero la lógica es la misma, nótese que no se utiliza una función recursiva, ya que eso es ineficiente para el desarrollo de aplicaciones en Android y que todos los datos no tienen accesores, si no que se invocan de manera directa. En la línea 140 se crea el nivel que se retornará en caso de que se encuentre, después en la línea 141 se impone una estructura iterativa que dejará salir solo sí el nivel es una hoja, el *for* de la línea 143 itera los nodos y permite así el control para escalar entre los diferentes nodos y hacer la búsqueda con respecto al parámetro "índice", sí no se encuentra entre los nodos la llave deseada, escalamos un nivel tal y como lo indica la sentencia de las líneas 151 a la 154, para así retornar el nivel propicio para buscar, insertar u eliminar.

```
88    /**
89     * Método alias para encontrar cualquier nodo en el árbol si existe
90     * @param indice
91     * @return
92     */
93    Nodo findNodo(int indice){
94        return (root.findNivelParaInsertar(indice)).findNodo(indice);
95    }
96
```

Figura 28. Alias para encontrar un nodo (Nivel.java)

La segunda parte representa el estado cuando una vez que ya encontramos el nivel con el que vamos a interactuar, ahora si podemos escudriñar los nodos. De igual manera se hace una

función alias en el archivo ArbolMainController.java como muestra la figura 26. En la cual dentro del nivel se hace una búsqueda entre los nodos, por el que se está buscando.

```
117
118    /**
119     * Función que encuentra un nodo a partir de su
120     * llave
121     **/
122    Nodo findNodo(int indice){
123        for (int i=0;i<nodos.size() ;i++ ){
124            Nodo nodo_temporal = nodos.elementAt(i);
125            if (nodo_temporal._indice > indice){
126                return null;
127            }
128            if (nodo_temporal._indice == indice)
129                return nodo_temporal;
130        }
131        return null;
132    }
```

Figura 29. Encontrando un nodo en el nivel (Nivel.java)

En el archivo Nivel.java se hace la segunda parte de la búsqueda dentro del nivel, apelando a las líneas 11-13 del algoritmo en el capítulo primero de la búsqueda, implementado con el parámetro índice y su búsqueda en las líneas 123-129, si no es encontrado regresamos un valor nulo 131 tal como en el algoritmo del capítulo primero en la línea 14.

En la implementación completa de la aplicación Android también se hizo la búsqueda por valor, que sigue el mismo patrón; encontramos el nivel dentro del árbol y posteriormente retornamos su valor.

```
112
113    /**
114     * Busca en la lista enlazada el rango que se desea
115     * @param indice_inferior
116     * @param indice_superior
117     * @return
118     */
119    String findValoresPorRango ( int indice_inferior , int indice_superior ){
120        return lista_doble.findRango( indice_inferior , indice_superior );
121    }
122
```

Figura 30. Método alias para búsqueda por rango (ListaDoble.java)

```
80     /**
81      * Regresa la colección de nodos
82      * en cuanto a un rango
83      */
84     public String findRango (int indice_inicial , int indice_final ){
85         String rc = "";
86         trackNodoInicial();
87         if (indice_final < indice_inicial ){
88             int tmp = indice_final;
89             indice_final = indice_inicial;
90             indice_inicial = tmp;
91         }
92
93          while ( Lista.Siguiente != null ){
94              if ( Lista._indice >= indice_inicial && Lista._indice <= indice_final ){
95                  rc += Lista.getDato() + " , ";
96              }
97              if ( Lista.Siguiente._indice > indice_final ){
98                  return rc;
99              }
100             //System.out.println( Lista._indice);
101             Lista = Lista.Siguiente;
102         }
103
104         if ( Lista._indice >= indice_inicial && Lista._indice <= indice_final ){
105                 rc += Lista.getDato() ;
106         }
107         return rc;
108    }
```

Figura 31. Función búsqueda por rango (ListaDoble.java)

La implementación de la búsqueda por rango se hizo diferente a la del algoritmo correspondiente al punto 1.5.2, ya que no se regresa un arreglo de nodos, sino meramente texto

para que Android lo pueda comprender y mostrarlo en la interfaz, de igual manera tiene su método alias en el archivo ArbolMainController.java; utilizando la variable de nuestra instancia que contiene a la lista doble y haciendo la búsqueda que apela al algoritmo de búsqueda por rango, como se muestra en la 39 44; en la que las líneas 87-90 nos reacomodan los índices de búsqueda para no buscar al revés y la implementación del algoritmo se refleja en las líneas 93-107 haciendo la búsqueda en la lista doble obedeciendo a su algoritmo del capítulo primero, sección algoritmos, capítulo búsqueda por rango, en las líneas 4-7 y concatenando los resultados de valores para su presentación en Android en formato String (cadena de texto).

5.1.13 Implementando el algoritmo de inserción

De igual manera asumimos que ya conocemos el manejo de intenciones y actividades en Android por lo que iremos directamente a la implementación, la interfaz Android arroja tanto la llave o índice y el valor para insertar, estos parámetros son tomados en la función alias en ArbolMainController.java como se muestra en la figura 32.

```
37
38    /**
39     * Metodo alias para insertar
40     **/
41    void insert(int indice , String valor){
42        if (indice != 0){ //Dado que 0 es nuestra referencia
43            Nodo nodo = findNodo(indice);
44            if (nodo == null){
45                Nivel nivel = findNivel(indice);
46                Nodo nuevo = new Nodo(indice);
47                nuevo.setValor ( valor );
48                nivel.insertNodo( nuevo );
49                Nodo lista = new Nodo ( indice );
50                lista.setValor (valor);
51                lista_doble.insertDatoOrdenado ( lista );
52            }
53        }
54    }
55
```

Figura 32. Función alias insertar (ArbolMainController.java)

En la línea 44 comprobamos que el índice no exista cumpliendo el prerrequisito descrito en la Tabla 5, acorde con el algoritmo descrito en el apartado 1.5.3, las líneas del algoritmo 1 y 2 se representan en la línea 45 al encontrar el nivel que vamos a utilizar, y las líneas 49-51 crean un nuevo nodo para insertarlo en la lista doble obedeciendo a la línea 11 del algoritmo.

```
161    void insert(Nodo node){
162        if (!nivel.isNivelLleno()){
163            nivel.addListaNodos(node);
164        }
```

Figura 33. Función insertar (NivelController.java)

De igual forma en el apartado 1.5.3, en las líneas del algoritmo 3 y 4 se representa en éste fragmento de código al insertar el nodo cuando el nivel tiene capacidad, tal como muestra la figura 33.

```
166                 nivel.addListaNodos(node);
167                 Nivel miembro_nivel=new Nivel(nivel.numero_maximo_nodos);
168
169                 Nodo nodo_medio=(nivel.nodos.elementAt(nivel.hoja_a_medias));
170                 int nodosize=nivel.nodos.size();
171                 for (int i=nivel.hoja_a_medias;i<nodosize ;i++ ){
172                     Nodo nodo_temporal=nivel.nodos.elementAt(i);
173                     miembro_nivel.addListaNodos(nodo_temporal);
174                 }
175                 for (int i=nivel.hoja_a_medias;i<nodosize ;i++ ){
176                     nivel.nodos.remove(nivel.hoja_a_medias);
177                 }
178                 if(nivel.isHoja()){
179                     Nodo nodo_temporal=new Nodo(nodo_medio._indice);
180                     //this.lastNode().setNivel(miembro_nivel);
181                     nodo_medio=nodo_temporal;
182                     nivel.linkToNodos(miembro_nivel);
183                 }else{
184                     miembro_nivel.primer_nodo.setNivel(nodo_medio.nivel_apuntado);
185                     miembro_nivel.nodos.remove(nodo_medio);
186                 }
187                 if (nivel.isRoot){
188                     nivel.createNivelRoot(nodo_medio);
189                 }else{
190                     nivel.padre.nivel.insertNodo(nodo_medio);
191                 }
192                 nodo_medio.setNivel(miembro_nivel);
```

Figura 34. Inserción cuando existe overflow (NivelController.java)

Las líneas del algoritmo 6-8 descrito en el apartado 1.5.3, denotan el procedimiento para cuando existe overflow, y están implementadas entre las líneas 166-186 de la figura 34 haciendo los copy-up's y/o push-up's correspondientes siguiendo las directrices que se analizaron en el capítulo 2. Para finalizar cuando el nivel es la raíz (líneas 187-192 en el algoritmo del primer capítulo) obedecen al algoritmo en cuanto a que la raíz se parte creando un nuevo nivel e insertando el nodo tal como se describe en las líneas 9 y10 del algoritmo descrito en el capítulo primero.

5.1.14 Implementando el algoritmo de eliminación

A sabiendas de cómo manejar actividades e intenciones en Android nos enfocamos al algoritmo exclusivamente; el algoritmo de eliminación en caso de que se quiera remover nodos de nuestro árbol es el que permite mantener balanceada la estructura de nuestro árbol; de igual manera tenemos su función alias en la clase ArbolNivelController.java que es como sigue;

```
56      /**
57       * Metodo alias para eliminar
58       */
59      void eliminar(int indice){
60          Nodo nodo = findNodo(indice);
61          if (nodo != null){
62              Nivel nivel = nodo.nivel;
63              nivel.deleteNodo(nodo);
64              lista_doble.eliminar(indice);
65          }
66      }
```

Figura 35. Método alias para eliminar (ArbolMainController.java)

En la figura 35, se muestra la función alias para poder eliminar un nodo, primeramente verificamos que exista (línea 61), entonces lo eliminamos de la estructura del árbol (línea 63) y de la lista enlazada (línea 64).

```
22  void delete(Nodo nodo){
23      if (this.nivel.isRoot){
24          this.nivel.deletedeLista(nodo);
25          if ( this.nivel.nodos.size() == 1 ){
26              this.nivel.changeRaizANivel(this.nivel.primer_nodo.nivel_apuntado);
27              this.nivel.isRoot=false;
28              this.nivel.primer_nodo.nivel_apuntado=null;
29          }
30          return;
31      }
```

Figura 36. Parte algoritmo de eliminación (NivelController.java)

En la figura 36 se implementa la descripción del capítulo primero, sección algoritmos, capítulo eliminación, en caso de que el nivel en la que se encuentra el nodo sea la raíz, eliminamos el nodo; y en caso de que el tamaño del nivel sea de un nodo se hace una restructuración (líneas 25-28).

```
46
47          Nivel nivel_reacomodo = nodo_padre_partido.nivel_apuntado;
48          if (nivel_reacomodo.isAMedias()){
49              if (nivel_reacomodo.isHoja()){
50                  if (numero_padre==0){
51                      for (int i=1;i<nivel_reacomodo.nodos.size() ;i++ ){
52                          Nodo nodo_temporal=(nivel_reacomodo.nodos.elementAt(i));
53                          this.nivel.addListaNodos(nodo_temporal);
54                      }
55
56                      this.nivel.setSiguiente(nivel_reacomodo.next);
57                      nivel_reacomodo.setSiguiente(null);
58                      nodo_padre.nivel.deleteNodo(nodo_padre_partido);
59                  }else{
60                      for (int i=1;i<this.nivel.nodos.size() ;i++ ){
61                          Nodo nodo_temporal=(this.nivel.nodos.elementAt(i));
62                          nivel_reacomodo.addListaNodos(nodo_temporal);
63                      }
64                      nivel_reacomodo.setSiguiente(this.nivel.next);
65                      this.nivel.setSiguiente(null);
66                      nodo_padre.nivel.deleteNodo(this.nivel.padre);
67                  }
```

Figura 37. Parte algoritmo eliminación (NivelController.java)

En la figura 37 tenemos el caso de cuando el nodo esta justamente a la mitad (línea 48) se presenta en 2 situaciones, cuando es el nivel hoja para hacer los reacomodos descritas en el algoritmo descrito en el capítulo segundo, sección Algoritmos, capítulo eliminación en las líneas 10 a la 12 cuando el nivel tiene espacio y es posible mezclar los nodos para que la fusión vote un nodo (líneas 57,65).

5.1.15 Implementación de la persistencia

5.1.15.1 Persistencia en nodos

Como bien muestra el diseño de la figura 38, se generó una clase de persistencia que permitiera hacer persistente el índice y cada nodo de manera separada en un archivo independiente, para así aprovechar las bondades de la serialización y al momento de levantar los nodos poder utilizar las propiedades del objeto y recorrerlo con una lista doblemente enlazada.

```
43⊖    public static void writeNodo( Nodo n ) throws IOException{
44             try {
45                 File root = new File(Environment.getExternalStorageDirectory(), "BPTree");
46                   if (!root.exists()) {
47                     root.mkdirs();
48                   }
49                 FileOutputStream archivo = new FileOutputStream( root.getAbsolutePath() + "BPTree"
50                                                                   + n._indice + ".dat" );
51                 ObjectOutput s = new ObjectOutputStream( archivo );
52                 s.writeObject(n);
53                 s.flush();
54                 archivo.close();
55                 root = null;
56                 archivo = null;
57                } catch (Exception e) {
58                   System.out.println (e);
59                }
60
61     }
62
```

Figura 38. Implementación de la persistencia por nodos serializandolos de manera independiente (Persistencia.java)

En la figura 39 podemos observar la implementación de la implementación por nodos, misma que está encapsulada por un try con una excepción de entrada y salida, para tener los archivos de los nodos de manera controlada se crea una carpeta, por lo que las líneas 45-47 nos dicen si existe dicha carpeta, posteriormente creamos un handler para el archivo en la línea 49 y para utilizar la serialización un ObjectOutput en la línea 51, procedemos a escribir el objeto, limpiar la memoria y cerrar el archivo, posteriormente hacemos nulos los objetos para limpiar la memoria, todo esto de las líneas 52-56, finalmente cachamos una excepción si existe algún error.

```
62⊖    public static Nodo readNodo ( int key ) throws IOException, Exception{
63         try {
64             File root = new File(Environment.getExternalStorageDirectory(), "BPTree");
65             if (!root.exists()) {
66                 root.mkdirs();
67             }
68             FileInputStream archivo    = new FileInputStream( root.getAbsolutePath() + key + ".dat" );
69             ObjectInputStream s = new ObjectInputStream( archivo );
70             Nodo n = (Nodo) s.readObject();
71             archivo.close();
72             root = null;
73             archivo = null;
74             return n;
75         } catch (Exception e) {
76             System.out.println (e);
77         }
78         return null;
79     }
```

Figura 39. Implementación de lectura de nodos serializados (Persistencia.java)

De igual manera en la figura 39, de las líneas 64-67 corroboramos la existencia del directorio para leer, si es factible, entonces de las líneas 68-74 levantamos el objeto de tipo Nodo y lo regresamos, sino, una excepción es lanzada en la línea 76, en caso de que no se pueda acceder regresamos un valor nulo “null” en la línea 78.

5.1.15.2 Persistencia en todo el árbol

En la figura 40 podemos vislumbrar que todo el árbol será serializado, por lo que para ésta implementación se implementaron dos funciones estáticas que levantan y bajan el árbol en la memoria ROM.

```
24⊖    public static void writeTree(ArbolMainController t){
25         try {
26             File root = new File(Environment.getExternalStorageDirectory(), "BPTree");
27             if (!root.exists()) {
28                 root.mkdirs();
29             }
30             FileOutputStream archivo = new FileOutputStream( root.getAbsolutePath() +  "tree.dat" );
31             ObjectOutput s = new ObjectOutputStream( archivo );
32             s.writeObject(t);
33             s.flush();
34             archivo.close();
35             root = null;
36             archivo = null;
37         } catch (Exception e) {
38             System.out.println (e);
39         }
```

Figura 40. Implementación de la serialización en todo el árbol

Se debe de notar que para ésta implementación a todas las clases que intervienen en el “engine” del árbol (Nodo, Nivel, Nivelcontroller, ListaDoble, ArbolMainController) se tiene que agregar en la definición de clase “implements Serializable”. La figura 40 muestra la serialización de un árbol, en sí es la misma que la de un nodo, sólo con la diferencia del parámetro de tipo ArbolMaincontroller, que es el que alberga toda la información y enlace de objetos del B+-Tree que se encuentra en memoria.

```
121⊖    public static ArbolMainController wakeupTree () throws IOException, Exception{
122         try {
123             File root = new File(Environment.getExternalStorageDirectory(), "BPTree");
124             if (!root.exists()) {
125                 root.mkdirs();
126             }
127             FileInputStream archivo   = new FileInputStream( root.getAbsolutePath() + "tree.dat" );
128             ObjectInputStream s = new ObjectInputStream( archivo );
129             ArbolMainController t = (ArbolMainController) s.readObject();
130             archivo.close();
131             root = null;
132             archivo = null;
133             return t;
134         } catch (Exception e) {
135             System.out.println (e);
136         }
137         return null;
138     }
139
140 }
```

Figura 41. Deserialización del B+-Tree (Persistencia.java)

La figura 41 nos muestra el despertar del árbol para cuando inicia la aplicación, si el archivo aún no existe simplemente se salta y se crea uno nuevo en memoria para posteriormente ser serializado.

Cabe hacer notar que los puntos de impacto en el modo de nodos serializables son en las funciones eliminar e insert del archivo ArbolMainController, mientras que para el modo de árbol

serializable, se hace al nivel de actividades al momento de insertar o eliminar o bien cuando se cierra/abre la aplicación.

5.2 Implementación por escritura de archivos

Es importante mencionar que toda la interface gráfica es la misma que se utilizó para las otras implementaciones, la diferencia radica en el "engine" que le da vida al B+-Tree, ya que todas las implementaciones poseen la misma interface al tener los métodos insertar, eliminar, buscar por índice, buscar por valor, o buscar por rango.

5.2.1 Clases

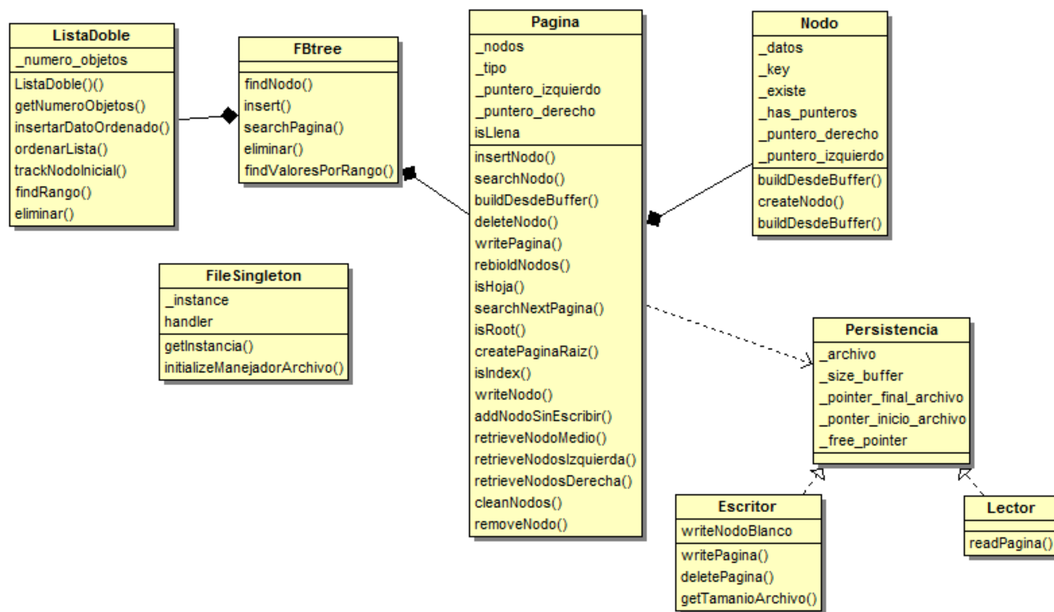


Figura 42. Diagrama de clases, implementación de persistencia a bajo nivel

El la figura 42, se re-diseña el B+-Tree para poder utilizarlo con persistencia a base de buffers y archivos, se utiliza una clase Persistencia que nos permite utilizar un Escritor y Lector de objetos página para poder escribir con bytes directamente cada campo de la página con la siguiente estructura:

Tipo Página	Puntero Página	Puntero Padre	Esta lleno	Índice	Contiene Punteros	Puntero Izquierdo	Puntero Derecho
Char	Long	Long	Short	Int	Short	Long	Long

Figura 43. Estructura del archivo de la persistencia

La figura 43 denota la estructura del archivo del cómo se escribe y lee desde el archivo índice por medio de buffers. Se debe de notar que una página está compuesta por 2 o más nodos, en la imagen se aprecia que la parte roja denota a toda la página, y la parte con los encabezados

negros representa un nodo, así pues el archivo podría quedar con múltiples nodos acorde al nivel de jerarquización que se desee implementar.

5.2.2 Implementación de la persistencia

Atendiendo a la persistencia se tiene un lector y escritor, ambos heredan de una clase padre que contiene la información del archivo en la que estos van a coexistir, así como los atributos mínimos necesarios para dar un funcionamiento óptimo, como podemos ver en la figura 44:

```
 5 public class Persistencia {
 6     private static String _archivo = "bptree.fab";
 7     public short _size_buffer = 144;
 8     public int _pointer_final_archivo = 0;
 9     public int _pointer_inicio_archivo = 0;
10     public int _pointer_libre = 0;
11     public RandomAccessFile btree_file;
12
13     public void __construct() throws IOException{
14         this.btree_file = new RandomAccessFile( Persistencia._archivo , "rw");
15     }
16 }
```

Figura 44. Clase Persistencia tercer_implementacion/Persistencia.java

Podemos observar que es el padre de un lector y de un escritor, contiene los atributos necesarios para poder manejarse dentro del archivo, así como también el tamaño del buffer que se utilizará para leer/escribir en el archivo, apelando al diseño de la figura 44. Ahora bien, utilizando el objeto página, descrito en el diseño de ésta implementación se puede utilizar el siguiente lector:

```
 5 public class Lector extends Persistencia {
 6     public Pagina readPagina( long numero_pagina ){
 7         Pagina p = new Pagina();
 8         FileSingleton manejador_archivo = FileSingleton.getInstance();
 9         try {
10             manejador_archivo.handler.seek( numero_pagina );
11             byte[] buffer = new byte[this._size_buffer];
12             manejador_archivo.handler.read( buffer );
13             ByteBuffer  buffer2 = ByteBuffer.wrap ( buffer );
14             p.rebuildNodos ( buffer2 );
15         } catch (IOException e) {
16          // TODO Auto-generated catch blocks
17          e.printStackTrace();
18         }
19         return p;
20     }
```

Figura 45. Lector en tercera_implementacion/Lector.java

El lector utiliza los atributos del objeto Pagina para poder leer desde el buffer una página completa, a partir de un *numero_pagina*, que representa la posición de la página que se desee leer en el archivo, para poder manipular el archivo en cualquier momento de la aplicación y no tener que crear manejadores individuales en la línea 10 se observa un manejador_archivo, mismo que apela a la implementación de un Singleton que contiene la información necesaria para acceder al archivo descrito por la clase persistencia como se muestra en la figura 45. Además el objeto Página

contiene una función que permite reconstruir los nodos a partir del flujo de bytes provenientes del archivo.

```
4  public class FileSingleton {
5      private static FileSingleton INSTANCE = new FileSingleton();
6      public RandomAccessFile handler;
7
8
9      private FileSingleton()  {
10         try {
11             this.initializeManejadorArchivo();
12         } catch (IOException e) {
13             e.printStackTrace();
14         }
15     }
16
17     private void initializeManejadorArchivo() throws IOException{
18             this.handler = new RandomAccessFile("aleatorio.txt", "rw");
19     }
20
21     public static FileSingleton getInstance() {
22         return INSTANCE;
23     }
24 }
25
```

Figura 46. Singleton para manejar archivo FileSingleton.java

```
50     public void rebuildNodos( ByteBuffer  buffer ){
51         this._tipo              = buffer.getChar();
52         this._puntero_pagina    = buffer.getLong();
53         this._puntero_padre     = buffer.getLong();
54
55         Vector<Nodo> v = new Vector<Nodo>(3);
56            for (int i = 0 ; i < 3 ; i++ ){
57                 Nodo n = Nodo.buildDesdeBuffer(buffer);
58                // System.out.print("Existe --> " + n.toString() );
59                   if ( n instanceof Nodo ){
60                       v.add( n );
61                   }
62            }
63         this._nodos = v;
64     }
```

Figura 47. Función para reconstruir Nodos a partir de un bufferPagina.java

Podemos observar además en la figura 47 que cada nodo se reconstruye individualmente, ya que el buffer se va recorriendo, por esto, la figura 56 muestra el método del nodo para poder consolidarse y después ser añadido al atributo _nodos de la página en la línea 60 de la figura 47.

En la figura 48 podemos observar como el nodo toma el buffer y lo lee acorde a sus atributos descritos en el diseño del archivo, esto con el fin de poder conformar la estructura de la página y poder armarla/desarmarla según la conveniencia del B+-Tree.

```java
public static Nodo buildDesdeBuffer( ByteBuffer buffer ){
    short existe = 0;
    existe       = buffer.getShort();

    if ( existe == 0 ){
        return null;
    }

    Nodo reconstruido = new Nodo();
    reconstruido._existe      = existe;
    reconstruido._key         = buffer.getInt();
    reconstruido._has_punteros= buffer.getShort();
    reconstruido.puntero_izquierdo_archivo = buffer.getLong();
    reconstruido.puntero_derecho_archivo   = buffer.getLong();
    return reconstruido;
}
```

Figura 48. Reconstrucción de un nodo por medio de buffer tercera_implementacion/Nodo.java

De igual forma el escritor emplea un objeto página para poder escribir los nodos y la información necesaria en el archivo apelando al diseño del archivo:

```java
4 public class Escritor extends Persistencia {
5     public boolean writePagina( Pagina p , long pos ) throws IOException{
6         boolean exito = true;
7         try {
8             FileSingleton manejador_archivo = FileSingleton.getInstance();
9             long longitud_archivo = manejador_archivo.handler.length();
10
11            if ( pos >= 0 ){
12                //Escribo donde quieras
13            } else {
14                p._puntero_pagina = longitud_archivo;
15                pos               = longitud_archivo;
16            }
17            //System.out.println ("POS --> " + pos );
18            manejador_archivo.handler.seek( pos );
19            manejador_archivo.handler.writeChar( p._tipo );
20            manejador_archivo.handler.writeLong( p._puntero_pagina );
21            manejador_archivo.handler.writeLong( p._puntero_padre );
22            System.out.println ("DA --> " + p._puntero_derecho + " " + p._puntero_izquierdo );
23
24                for (int i = 0 ; i < p._nodos.size() ; i++ ){
25                    manejador_archivo.handler.writeShort( p._nodos.elementAt(i)._existe );
26                    manejador_archivo.handler.writeInt( p._nodos.elementAt(i)._key );
27                    manejador_archivo.handler.writeShort( p._nodos.elementAt(i)._has_punteros );
28                    manejador_archivo.handler.writeLong( p._nodos.elementAt(i).puntero_izquierdo_archivo );
29                    manejador_archivo.handler.writeLong( p._nodos.elementAt(i).puntero_derecho_archivo );
30                }
31                short numero = 0;
32                for (int i = 0 ; i < (3 - p._nodos.size()) ; i ++){
33                    Escritor.writeNodoBlanco( numero , 0, numero, 0, 0);
34                }
35
36        } catch (IOException e) {
37            exito = false;
38            e.printStackTrace();
39        }
40        return exito;
41    }
```

Figura 49. Escritor tercer_implementacion/Escritor.java

Como se puede apreciar en la figura 49, en la línea 8 se utiliza un patrón de diseño llamado *singleton* para manejar el archivo, los parámetros que recibe esta función son la página y el lugar en el archivo que le corresponderá, si la página es nueva, entonces se condiciona de las líneas 11 a la 16, para obtener la posición de la página y sobrescribir o anexarla como nueva al archivo. Posteriormente escribimos la información vital para la página (lo rojo de la figura 43) en las líneas 19-21, y posteriormente la información de los nodos que contenga la página, líneas 24-34, se escriben nodos en blanco para solo rescribir valores intercambiables.

Ahora, las funciones que se implementaron para el B+-Tree se muestran a continuación, primero la función primordial que arroja la búsqueda de la página en la que vamos a interactuar:

5.2.3 Implementación de búsqueda

```
6    public static Pagina searchPagina ( int key ){
7        //Inicializamos la página root.
8        Pagina p = null;
9        Lector l = new Lector();
10       p = l.readPagina(0);
11       //p.print();
12       while ( p != null ){
13           if ( p.isHoja() ){
14               //n = p.searchNodo( key );
15               return p;
16           } else {
17               long puntero_siguiente = p.searchNextPagina( key );
18               //System.out.print("PS --> " + puntero_siguiente );
19               if (puntero_siguiente > 0){
20                 p = l.readPagina( puntero_siguiente );
21               } else {
22                 return null;
23               }
24           }
25       }
26       return null;
27   }
```

Figura 50. Algoritmo búsqueda tercer_implementacion/FBTree.java

En la figura 50 podemos ver reflejado el algoritmo de búsqueda que se describe en la sección de algoritmos en el capítulo primero, utilizando objetos página y un lector para buscar la página acorde a un *key*, y regresar la página en la cual se encuentra. Se itera hasta que la página sea *null*, o bien hasta que la página en la que se encuentre sea una página de tipo hoja. En la línea 17 se observa una función que busca la siguiente página para seguir iterando y bajando por el árbol, esa función se muestra en la figura 51:

```
97   public long searchNextPagina( int key ){
98       long puntero = -1;
99       Nodo elegido = null;
100      for ( int i=0; i < this._nodos.size() ; i++ ){
101          if ( this._nodos.elementAt(i)._key  > key ){
102             elegido = this._nodos.elementAt(i);
103             break;
104          }
105      }
106
107      if ( this._nodos.size() > 0 ){
108          if (elegido == null){
109              elegido = this._nodos.lastElement();
110          }
111
112          if ( elegido._has_punteros == 1 ){
113            puntero = ( key >= elegido._key )? elegido.puntero_derecho_archivo
114                                             : elegido.puntero_izquierdo_archivo;
115          } else {
116          }
117     }
118      return puntero;
119  }
120
```

Figura 51. Función que busca la siguiente página tercer_implementacion/Pagina.java

De ésta manera podemos iterar los nodos guiándose por el parámetro *key*, y poder ir leyendo entre las páginas la idónea a manera de buscar en el árbol, tal y como se explicó en el capítulo primero, en la sección de búsqueda. Acompañado de éste algoritmo una vez que se tiene la página, se puede buscar una coincidencia para buscar una llave exacta también, en la figura 52 se muestra la función que el objeto página tiene para encontrar un nodo dentro de sí misma, con esto se completa la búsqueda por página y la del nodo dentro de la misma.

```
88    public Nodo searchNodo( int key ){
89        for ( int i=0; i < this._nodos.size() ; i++ ){
90            if ( this._nodos.elementAt(i)._key  == key ){
91                return this._nodos.elementAt(i);
92            }
93        }
94        return null;
95    }
96
```

Figura 52. Buscar Nodo (Pagina.java)

5.2.4 Implementación de inserción

El algoritmo de Inserción se apoya en las funciones anteriores para poder saber si ya existe un nodo con una llave en el árbol y así mismo encontrar la página correspondiente en la cual insertar el nuevo nodo, o bien desde la cual se debe de ir haciendo overflow hasta consolidar una nueva estructura para el árbol, como podemos apreciar en la figura 53:

```
29    public void insertNodo ( int key , String datos ) throws IOException{
30        Nodo    n    = Nodo.createNodo(key, datos);
31        Pagina p    = FBTree.searchPagina( key );
32        Lector l    = new Lector();
33        Escritor e = new Escritor();
34        if (p == null){
35            p = Pagina.createPaginaRaiz();
36            p.writeNodo(n);
37        } else {
38            if ( p.searchNodo(key) == null ){
39                if ( p.isLlena() ){
40                    while ( p.isLlena() ){
41                        Pagina  p_izquierda = new Pagina();
42                        Pagina  p_derecha   = new Pagina();
43                        p.addNodoSinEscribir( n );
44                        p_izquierda.addNodoSinEscribir( p.retrieveNodosIzquierda() );
45                        p_derecha.addNodoSinEscribir( p.retrieveNodosDerecha() );
46                        if ( p.isHoja() ){//Copy up
47                           p_derecha.addNodoSinEscribir( p.retrieveNodoMedio() );
48                           p_izquierda._tipo = 'D';
49                           p_derecha._tipo   = 'D';
50                        } else {//Push up
51                           p_izquierda._tipo = 'I';
52                           p_derecha._tipo   = 'I';
53                        }
54                       p._tipo = 'I';
55                       Nodo temporal = p.retrieveNodoMedio();
56                       if ( p.isLlena() );
57                         p.cleanNodos();
58                       p_izquierda._puntero_padre = p._puntero_pagina;
59                       p_derecha._puntero_padre   = p._puntero_pagina;
60                       p_izquierda.writePagina( -1 );
61                       p_derecha.writePagina( -1 );
62                       temporal.puntero_izquierdo_archivo   = p_izquierda._puntero_pagina;
63                       temporal.puntero_derecho_archivo     = p_derecha._puntero_pagina;
64                       temporal._has_punteros               = 1;
65                       p.addNodoSinEscribir( temporal );
66                       p.writePagina( p._puntero_pagina );
67                    }
68                } else {
69                  p.writeNodo(n);
70                }
71            } else {
72                System.out.println ("El indice" + key + "ya ta" );
73            }
74        }
```

Figura 53. Inserción B+-Tree FBTree.java

Podemos contemplar que inicializamos nuestro nodo nuevo, página, lector y escritor, líneas 30-33, las cuales serán nuestras herramientas a lo largo de la función, inmediatamente si la página devuelta es nula, sabemos que no existe un árbol y que debemos de crearlo, por lo que en las líneas 34-37 se crea, si no es el caso, debemos ver si la página está llena, en la condición de la línea 39, en caso de que no esté llena, simplemente insertamos, línea 69, caso contrario, debemos hacer un overflow haciendo copy-up y push-up (previamente explicados en el capítulo primero, sección algoritmos, parte inserción) hasta llegar al nodo root, por lo que se utiliza el *while* en la

línea 40 y se crean las páginas para separar los nodos de las líneas 40-41, posteriormente se evalúa la situación de la página, es hoja, índice o root, para mover los punteros acorde al caso, así como también los tipos de las páginas.

5.2.5 Implementación de la eliminación

```
 81
 82      public void delete ( int key ){
 83             Nodo    n = Nodo.createNodo( key, val );
 84             Pagina p = FBTree.searchPagina( key );
 85
 86               if (p.searchNodo(key)){
 87                   p.removeNodo(key);
 88                   while ( !p.isLlena() ){
 89                        Pagina bucket = new Pagina();
 90                        Nodo t = p.getNodoMedio();
 91                        bucket._nodos = p.getNodosIzquierdaDerecha(true);
 92                        p._nodos = p.getNodosIzquierdaDerecha(false);
 93                        t.cleanValor(); t._puntero_derecho    =  bucket;
 94                        t._puntero_izquierdo =   p;
 95                        switch ( p._tipo ){
 96                        case 'O': Pagina nueva_raiz    =  new Pagina();
 97                                  nueva_raiz.addNodo(t); this._root = nueva_raiz;
 98                                  bucket._padre = this._root;
 99                                  bucket._tipo  = 'D'; p._padre      = this._root;
100                                  p._tipo       = 'D';
101                                  break;
102                        case 'R': Pagina ri    =  new Pagina();
103                                  ri.addNodo(t); this._root = ri;
104                                  bucket._nodos.removeElementAt(0);
105                                  bucket._padre = this._root;
106                                  bucket._tipo  = 'I';  p._padre      = this._root;
107                                  p._tipo       = 'I';
108                                  break;
109                        case 'D':
110                                  p._padre.addNodo( t );
111                                  bucket._tipo  = 'D'; bucket._padre = p._padre;
112                                  break;
113                        case 'I': bucket._nodos.removeElementAt(0);
114                                  bucket._padre = p._padre;
115                                  break;
116                        }
117                        p = p._padre;
118                        if ( p._padre == null ){
119                            p._tipo = 'R';
120                        }
121                        if (p == null)
122                            return;
123                   }
124               } else {
125                   //No existe el nodo
126               }
```

Figura 54. Implementación de la eliminación FBTree.java

La figura 54 denota que para el algoritmo de eliminación primeramente tenemos que tener el nodo y la página en la que se encuentra, líneas 83-84, posteriormente se tiene que eliminar de la página, línea 87, y posteriormente ver si existe un underflow, si es el caso la página no está llena, por lo que debemos de repartir y fundir los nodos, líneas 91 a 94, posteriormente según el tipo de la página se ajustan los punteros y repetimos el proceso.

Como puntos a resaltar en éste capítulo cabe hacer notar que la persistencia de más fácil implementación es la serialización, preferentemente la que serializa todo el árbol, ya que no se requiere clases especializadas debido a que son parte del motor de java.

5.3 Utilización del árbol

Hasta este punto se tienen tres implementaciones del árbol, las cuales representan bibliotecas que utilizan una misma interface para poder funcionar. Tomando ventaja de esto podemos describir la forma en la que se utiliza cualquier implementación del árbol para poder utilizarlo.

Para las implementaciones basadas en serialización basta con obtener una instancia por medio del *Singleton:*

ArbolMainController bptree = ArbolMainController::getInstancia()

Mientras que para la implementación con archivos, tenemos que crear una instancia:

FBTree bptree = new FBTree();

Una vez que se tiene la instancia, podemos hacer uso de la interfaz que comparten todas las implementaciones para las operaciones básicas:

5.3.1 Inserción

Insertará en la estructura del árbol la llave 35 con el valor "El dato String que deseemos". En la implementación de nodos serializados crea un archivo con el objeto Nodo que contiene la información.

Bptree.insert(35,"El dato String que deseemos");

5.3.2 Eliminación

De encontrarse el nodo en el árbol lo eliminará tanto de la lista doble enlazada como de la estructura del árbol, en la implementación de nodos serializados elimina el archivo que se crea.

Bptree.eliminar(35);

5.3.3 Búsqueda

Devuelve un objeto Nodo, con la llave 35, de este modo podemos acceder a su atributo *_datos* para obtener su información.

Bptree.findNodo(35);

5.3.4 Búsqueda por valor

Consulta la lista enlazada, iterando los nodos y comparando la cadena dada con los valores de los nodos, a fin de encontrar la coincidencia y regresar el Nodo correspondiente.

Bptree.findNodoPorValor("Encontrar nodo");

5.3.5 Búsqueda por rango

Con la primer llave (35) realiza una búsqueda normal, como la descrita en el punto 5.3.3, después se ubica el nodo en la lista doble enlazada y se regresan los nodos siguientes o anteriores, según el segundo parámetro.

Bptree.findValoresPorRango(35,48);

Ahora bien, con los métodos anteriores cada uno de las implementaciones funciona de diferente manera, esto es: serializando los nodos hoja, escribiendo en archivos o bien serializando el árbol en su totalidad, para ejemplificar el uso de la biblioteca, creamos un conjunto de XML auxiliares que sirven para proveer una interfaz gráfica y así poder ser accedida por usuarios.

5.3.6 Diseño de pantallas compartidas

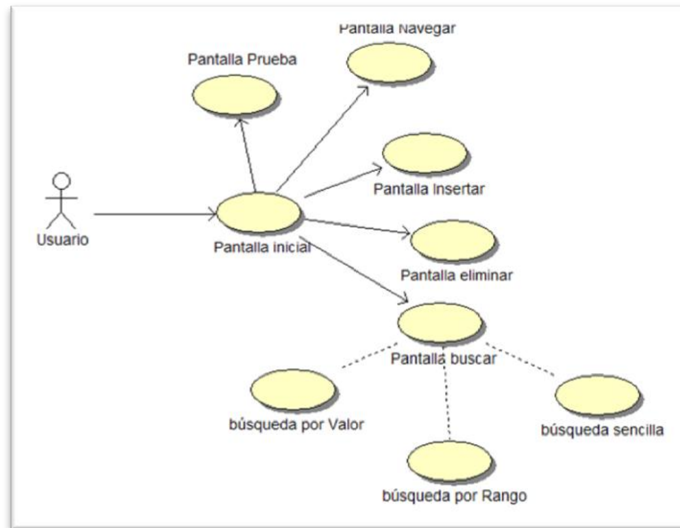


Figura 55. Diagrama de casos de uso B+-Tree Android

En la figura 55 podemos analizar un diagrama de casos de uso, que oferta un abanico de opciones que tendrá nuestra aplicación Android, en seguida mostraremos las interfaces que se tendrán apelando a cada caso de uso, podremos utilizar la biblioteca para cada función básica:

Figura 57. Pantalla principal

- Contiene los botones insertar, buscar y eliminar para modificar al antojo la estructura del árbol.
- Contiene un botón de navegar, para ver la estructura del árbol.
- Contiene un “Acerca de..” describiendo el uso de la aplicación, su autor y sede.

- Contiene un campo de texto para ingresar de manera manual el índice o llave.
- Contiene un campo de texto para ingresar el valor que deseamos que contenga esa llave.
- Contiene un botón para ejecutar la acción de inserción.
- Apela al punto 5.3.1.

Figura 58. Pantalla de inserción

Figura 59. Pantalla de eliminación

- Contiene un campo de texto para ingresar de manera manual el índice o llave que vamos a eliminar.
- Contiene un botón para ejecutar la acción de Eliminar.
- Apela al punto 5.3.2.

- Cuando se pulse el botón de buscar nos aparecerá un menú emergente; el cual contiene las opciones de búsqueda: por índice, por valor o por rango.

Figura 60. Pantalla selección método de búsqueda

Figura 61. Pantalla de búsqueda por índice

- Contiene un campo de texto para ingresar el índice o la llave
- Contiene un botón de buscar que ejecuta la acción de búsqueda en el árbol B+-Tree.
- Contiene un campo de resultados para mostrar las coincidencias.
- Contiene un botón para regresar a la actividad principal.
- Apela al punto 5.3.3.

- Contiene un campo para ingresar el valor a ser buscado.
- Contiene un botón de buscar que ejecuta la acción de búsqueda en el árbol B+-Tree
- Contiene un campo de resultados para mostrar las coincidencias.
- Contiene un botón para regresar a la actividad principal.
- Apela al punto 5.3.4.

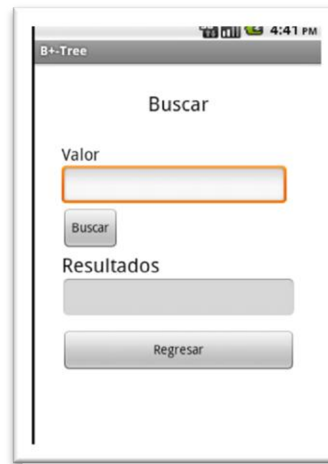


Figura 62. Búsqueda por valor

Figura 63. Búsqueda por rango

- Contiene un campo de texto para ingresar el índice inicial
- Contiene un campo de texto para ingresar el índice final
- Contiene un botón de buscar que ejecuta la acción de búsqueda en el árbol B+-Tree.
- Contiene un campo de resultados para mostrar las coincidencias.
- Contiene un botón para regresar a la actividad principal.
- Apela al punto 5.3.5.

6. Capítulo Sexto: Evaluación del Performance

En el presente capítulo se muestra la evaluación de las tres implementaciones realizadas en el capítulo anterior y las compara contra la implementación comercial de Virtual Machinery, a fin de ver las debilidades y fortalezas.

La motivación es la de comparar los resultados de la investigación que se realizó contra una implementación comercial, a fin de ver en qué operaciones es más eficiente y donde se podría optimizar.

El capítulo se dividirá en cuatro apartados: El primero definirá cómo es que se obtuvieron los conjunto de datos a fin de realizar las pruebas en las diversas implementaciones; el segundo se enfoca a la velocidad con la que se realizan las operaciones; el tercero muestra la utilización de la memoria del dispositivo; y el cuarto presenta métricas de espacio respecto del tamaño en disco.

Se debe de considerar que todas las pruebas fueron realizadas en un ambiente Android con un API 2.2, en la que se configuró un emulador estándar con procesador ARM (armeabi), una memoria interna de 8Gb y una memoria SD de 2 GB. Todo esto utilizando la herramienta de Motorola (MotodevStudio) y el SDK de Android 2.0, sobre una máquina con Windows 7, con 6Gb de RAM y un procesador i5-480M. Las razones de utilizar Android como plataforma es por que es gratuito y actualmente es el sistema operativo que tiene tendencia en millones de dispositivos, superando al iOS, con respecto a los SmartPhone, además no se necesita de pagar por una licencia para poder utilizarlo.

6.1 Datos de prueba para implementaciones

Para realizar una comparación entre diversas implementaciones de una manera justa, es evidente que se debe de tener los mismos conjuntos de datos: por lo que se observó el funcionamiento de la implementación de Virtual Machinery, a fin de replicar el comportamiento a través de un generador de datos.

5.1.1 Generador de datos

El generador de datos realizado es un objeto que nos permite obtener registros aleatorios a partir de un conjunto de arreglos que contienen nombres, apellidos, ciudades, áreas y demás, utilizados de manera aleatoria para crear registros *dummy* para hacer pruebas. La figura 64 muestra el código que nos permite utilizar los diferentes arreglos para obtener un registro:

```
58⊖    public String generateRegistro(){
59         StringBuffer localStringBuffer = new StringBuffer();
60          localStringBuffer.append(this._nombres[this.randomGenerator.nextInt(9)]);
61          localStringBuffer.append(" ");
62          localStringBuffer.append(this._apellidos[this.randomGenerator.nextInt(11)]);
63          localStringBuffer.append(" ,");
64          localStringBuffer.append(1 + this.randomGenerator.nextInt(59));
65          localStringBuffer.append(" ");
66          localStringBuffer.append(this._ciudades[this.randomGenerator.nextInt(4+1)]);
67          localStringBuffer.append(" ");
68          localStringBuffer.append(this._tipos_calle[this.randomGenerator.nextInt(4)]);
69          localStringBuffer.append(" ,");
70          localStringBuffer.append(this._ciudades[this.randomGenerator.nextInt(4)]);
71          localStringBuffer.append(" ,");
72          localStringBuffer.append(this._paises[this.randomGenerator.nextInt(4)]);
73          return localStringBuffer.toString();
74     }
```

Figura 64. Función que genera un registro aleatorio (DataGenerator.java)

El generador de datos se realizó para darle un equilibrio a las pruebas, utilizando las mismas longitudes de cadena (no mayores a 150 caracteres). Para lo que resta del proceso de la evaluación de las pruebas, procederemos a describir los conjuntos de datos que se utilizarán:

Conjunto de Datos	Descripción
S10	El conjunto de datos comprende de 10 registros.
S100	El conjunto de datos comprende de 100 registros.
S1000	El conjunto de datos comprende de 1000 registros.
S10000	El conjunto de datos comprende de 10000 registros.

Tabla 21. Conjunto de datos de pruebas

Cada uno de los registros retorna una cadena que no excede los 150 caracteres, dicho registro es asociado con la llave correspondiente dentro del B+-Tree, tomando en cuenta que el cómputo de lo que se hace con el registro entre cada implementación es independiente.

6.2 Tiempos de ejecución en las operaciones del B+-Tree

Las siguientes pruebas demuestran los resultados del performance del B+-Tree en sus diferentes implementaciones, analizaremos las pruebas de velocidad en inserción, consulta y eliminación, con el objetivo de evaluar el performance de las diferentes implementaciones.

Cabe hacer notar que en algunas pruebas del árbol B+-Tree con nodos Serializados no se pudieron obtener los datos debido a un error que marcaba del flujo de información, debido al estrés de generar múltiples archivos en un dispositivo móvil, el problema trató de resolverse por medio de hilos, mejorando la creación de archivos hasta los 100 registros, sin embargo, con pruebas más grandes no se pudo realizar las pruebas.

6.2.1 Velocidad de inserción

La tabla 22 muestra una comparativa en cuanto a la operación de inserción realizada en las diversas implementaciones, se utilizan los conjuntos de datos mencionados en la sección 6.1.

Operación	Conjunto de Datos	Virtual Machinery	B+-Tree Serializado (Tiempo de inserción + Tiempo Serialización del árbol)	B+-Tree Acceso Aleatorio	B+-Tree Nodos Serializados
Inserción	S100	225	95+40	384	1860
Inserción	S1000	1393	1056+71	1934	No disponible
Inserción	S10000	8603	6024+71	13458	No disponible
Inserción	S100000	154225	125846+82	182348	No disponible

Tabla 22. Comparación en operación de inserción (Unidades en milisegundos)

En base a la tabla 22 se realizó una gráfica mostrada en la figura 65, en la que se puede ver que el B+-Tree Serializado es más rápido al momento de hacer inserciones, para la gráfica se contrasta la implementación comercial en contraste con la mejor implementación de las tres expuestas en el documento (B+-Tree Serializado). Se debe considerar que éste tiempo se obtiene sumándole lo que el árbol se tarda en serializarse, tiempo que no debe incluirse mientras se utiliza la aplicación. Entonces debe quedar asentado que para levantar/guardar el árbol sólo se ejecuta al inicio/fin de la aplicación, por lo que no debe tomarse el tiempo de serialización tras cada inserción. Una de las grandes ventajas que se tiene es que la serialización vincula un canal con el objeto en memoria y el archivo [Lee], por lo que el proceso de guardado es más eficiente, debido a la relación que el objeto en memoria y el archivo serializado poseen.

Para la gráfica mostrada en la figura 66, se toman encuentra solo las mejores implementaciones B+-Tree Serializado y la implementación comercial del Virtual Machinery.

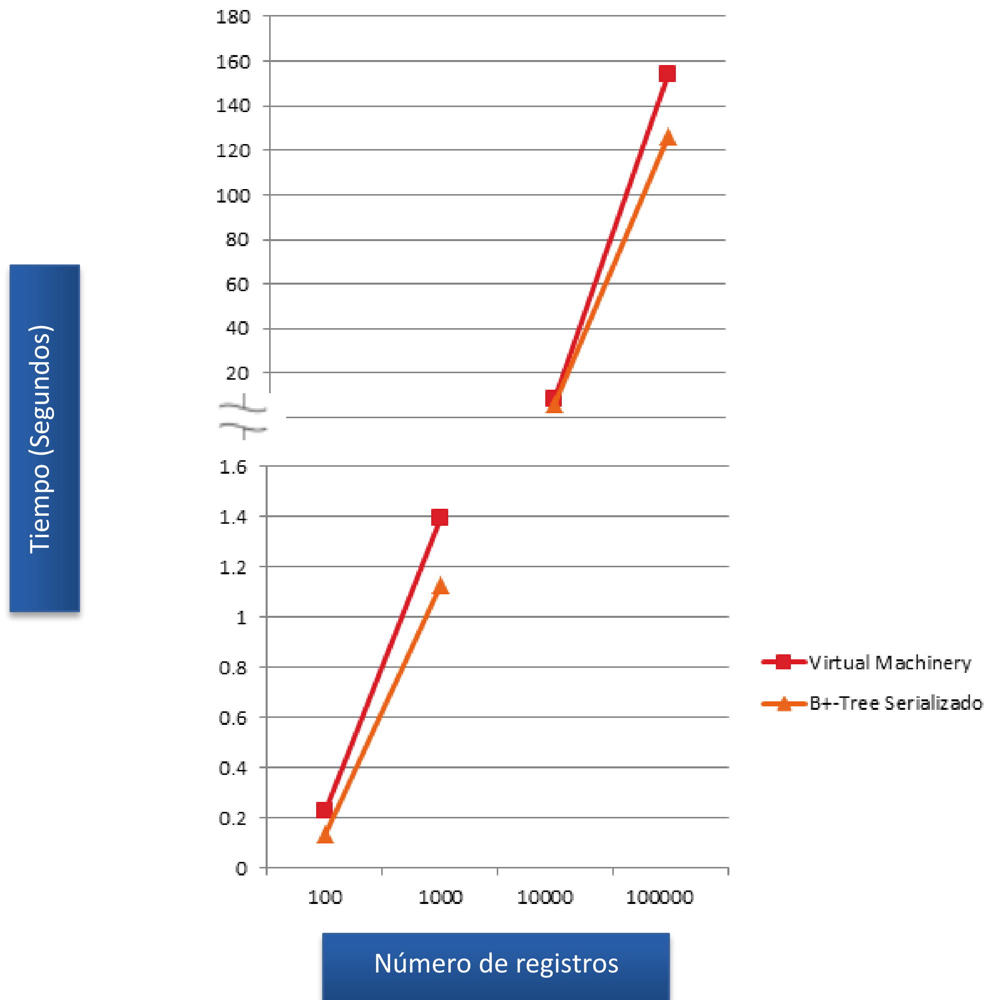


Figura 65. Comparación en operación de inserción

Podemos contrastar las mejores implementaciones, pudiendo observar que el tiempo que el B+-Tree serializado baja con respecto del Virtual Machinery en promedio es de un 26.64%,, con respecto a los resultados de todas las pruebas realizadas, cabe hacer notar que este porcentaje es tomado en consideración para ser justos cuando los datos son persistentes, esto es, cuando el archivo se guarda, debido a que el B+-Tree Serializado permanece en memoria y no guarda cada vez que se insertan datos, sino hasta que la aplicación se cierra.

6.2.2 Velocidad de consulta

Utilizando los conjuntos de datos descritos en 6.1, en la tabla 23 se muestra una comparativa en base a la operación de inserción en las diferentes implementaciones.

Nuevamente se hace énfasis que los datos no obtenidos por parte de la implementación de nodos serializados son debido al error de flujo en los archivos que se presentó en la plataforma Android.

Operación	Conjunto de Datos	Virtual Machinery	B+-Tree Serializado (Tiempo de inserción + Tiempo Serialización del árbol)	B+-Tree Acceso Aleatorio	B+-Tree Nodos Serializados
Consulta	S100	3230	1124	4895	6389
Consulta	S1000	37775	19875	52897	No disponible
Consulta	S10000	1116441	428495	1965487	No disponible
Consulta	S100000	43285452	32845658	87895635	No disponible

Tabla 23. Comparación en operación de búsqueda (Unidades en Milisegundos)

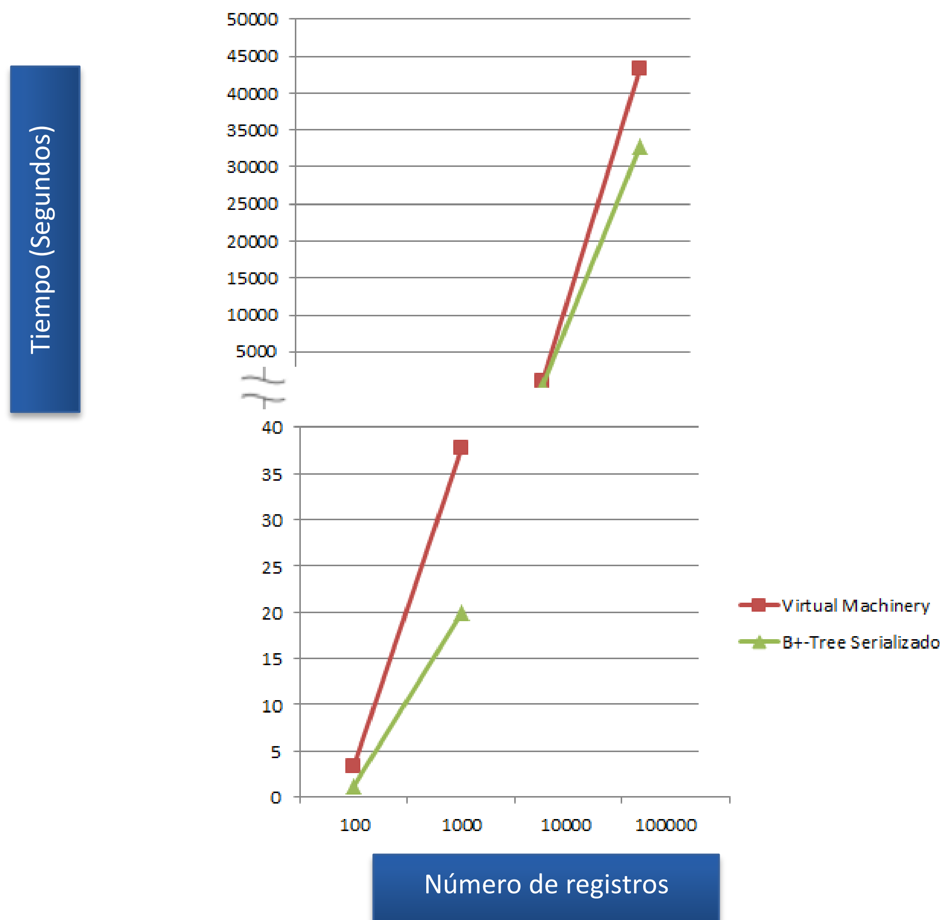


Figura 66. Comparación en operación consulta

Interpretando las cifras obtenemos un 47% de mejora promediando los resultados de todas las pruebas realizadas, ya que los datos son procesados en memoria RAM, mientras que los datos del Virtual Machinery son consultados exhaustivamente en disco, por una parte el resultado es bueno, pero a medida que el número de datos crece el rendimiento del dispositivo se ve afectado, por eso en la sección de trabajo futuro se hace énfasis a manejar la persistencia del árbol en varios archivos, para no sobrecargar tanto la memoria, los resultados tienen un mejor desempeño cuando se utilizan conjuntos de datos pequeños.

6.2.3 Velocidad de eliminación

En la tabla 24 se muestra una comparativa en las operaciones de eliminación en base al tiempo, en ésta prueba, la implementación comercial en su demo de Android no permite obtener los tiempos de eliminación, por lo que se tomaron de su página oficial en base a los resultados publicados en un experimento realizado bajo un Windows XP con un procesador de 2 GHz, la página no expone el dato de la cantidad de memoria RAM utilizada, y nos dice que para hacer una eliminación en promedio tarda 1.895 ms [VirtualMachinery 2012] por registro, por lo que se tiene en mente que ésta comparación es injusta al no ser bajo las mismas condiciones y bajo la misma plataforma.

Operación	Conjunto de Datos	Virtual Machinery	B+-Tree Serializado (Tiempo de inserción + Tiempo Serialización del árbol)	B+-Tree Acceso Aleatorio	B+-Tree Nodos Serializados
Eliminación	S100	130.8	1136+32	5124	10042
Eliminación	S1000	130800	19887+83	61492	No disponible
Eliminación	S10000	1308000000	428507+145	2158943	No disponible

Tabla 24. Comparación en operación de eliminación

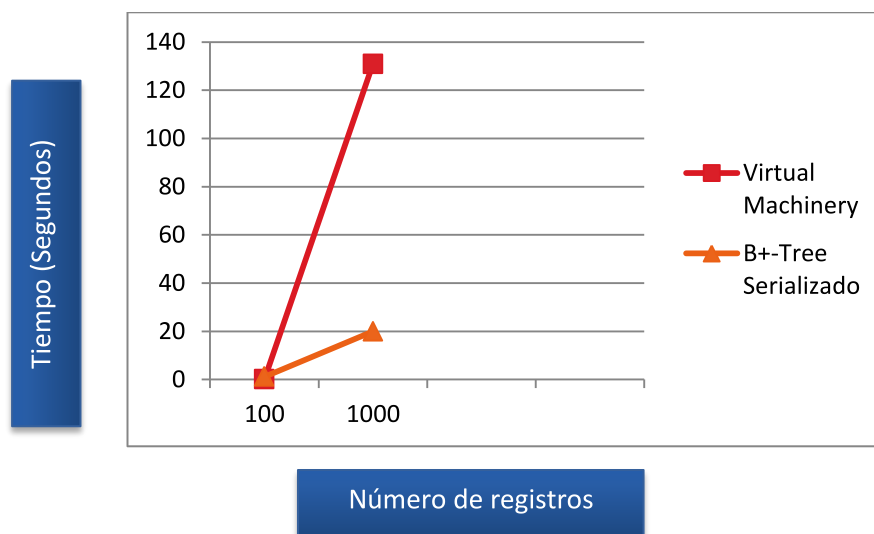


Figura 67. Comparación en operación de eliminación

En la figura 67, podemos observar que el B+-Tree es eficiente, pero se reitera que no hay una comparación justa, debido a que se toma una cantidad de tiempo fija para el Virtual Machinery, que fue provista desde su sitio de internet.

6.2.4 Capturas de Pantalla representativas en caso de inserción

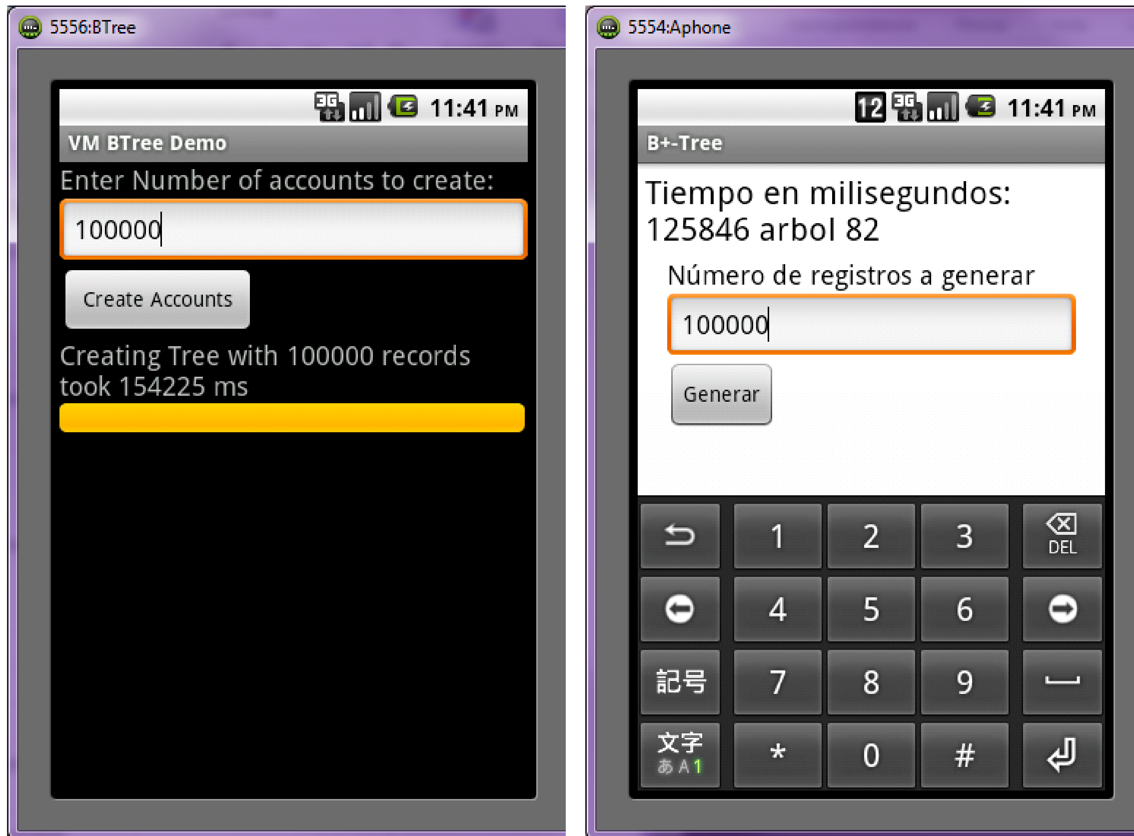


Figura 68. Capturas de pruebas de inserción

La figura 68 muestra capturas de inserción con 100,000 registros, del lado izquierdo se tiene al Virtual Machinery, y del derecho al B+-Tree Balanceado.

6.3 Recursos del dispositivo (Memoria)

No basta comparar las estadísticas de tiempo, habría que ver el comportamiento propio del dispositivo, ver cuánta memoria utiliza, ver cuánto poder de procesamiento requiere. Por lo que vamos a analizar primeramente los recursos de la pila de memoria, cabe hacer notar que las pruebas de la memoria fueron tomadas después de una inserción de 1000 registros, utilizando los set de datos S1000:

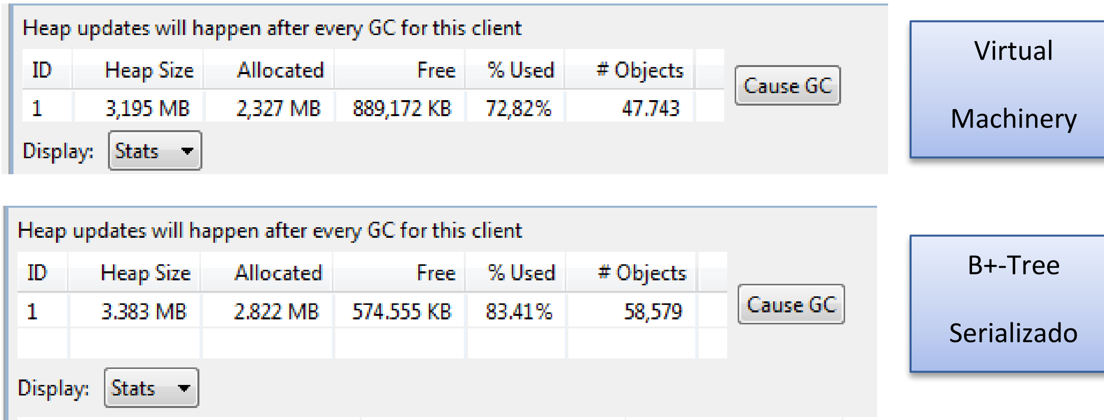
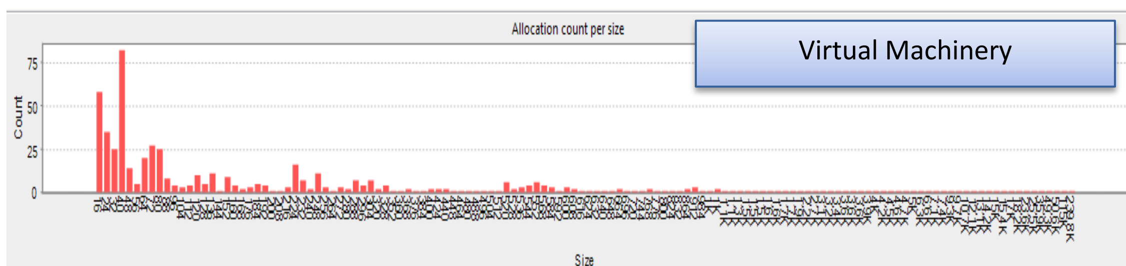


Figura 69. Rendimiento de Memoria B+-Tree Serializado Vs Virtual Machinery

Se puede observar en la figura 69, que el tamaño de la pila es rebasado por el B+-Tree en un 6%, lo cuál es muy bueno, ya que se llega a un equilibrio de no más del 10% en la pila y además se conserva extremadamente rápido el algoritmo de B+-Tree serializado una vez que está en memoria, lo que permite trabajar más rápido y serializar sólo lo necesario. Las métricas son arrojadas cada vez que el colector de basura es invocado, para obtenerlas forzamos a la herramienta (MOTODEV studio) a que nos diera las estadísticas inmediatamente después de la inserción de los 1000 registros, en la que el Virtual Machinery tardó en generarlos alrededor de 1397 ms, mientras que el B+-Tree Serializado tomó un tiempo de 1158 ms (incluyendo la serialización del árbol) éstas dos últimas resultaron de esta prueba. Al pagar el 6% de diferencia en la pila de memoria excedente del B+-Tree, se obtiene un rendimiento mayor al realizar operaciones, tal como se muestra en el apartado 6.2, reflejando una optimización importante y demostrando ser más efectivo, ya que fue programado tomando en cuenta las consideraciones de una aplicación para Android descritas en el apartado 3.4.1.



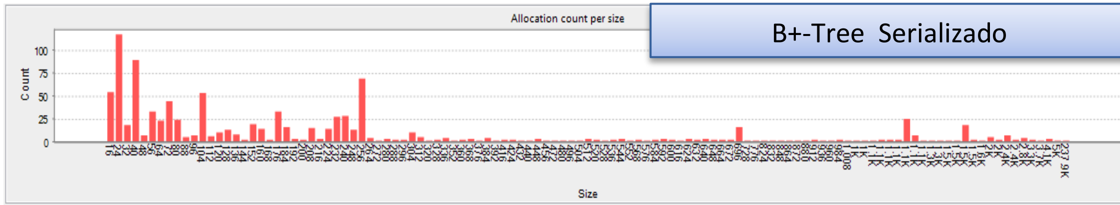


Figura 70. Comparación de Memoria Libre

En la figura 70, se puede observar que hay más utilización de memoria por el B+-Tree Serializado a lo largo del tiempo, ya que hace sus inserciones en memoria, por lo que Virtual Machinery tiene un 22% más de memoria libre, revelando que nuestra implementación es óptima y rápida para un conjunto de datos pequeños.

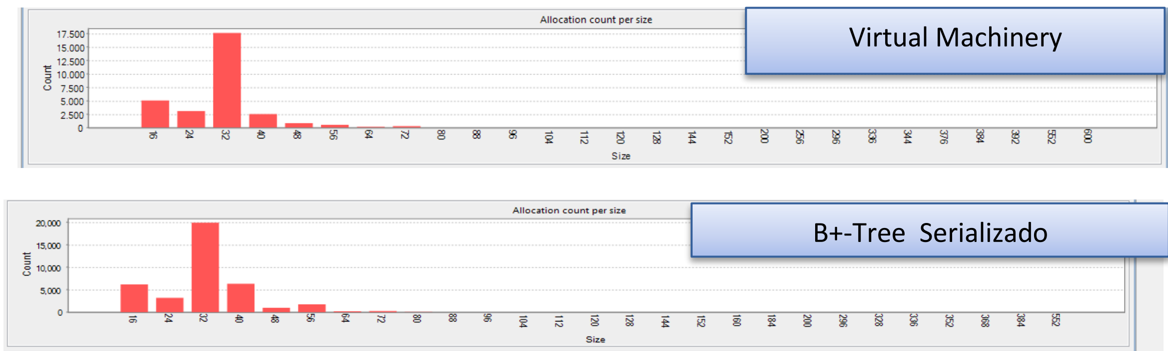


Figura 71. Comparación de los objetos utilizados

De igual manera existe un incremento que el algoritmo B+-Tree paga en objetos que guardan datos en memoria a cambio del performance que ofrece una vez que se encuentra en memoria, tal y como se muestra en la figura 71.

Virtual Machinery

Type	Count	Total Size	Smallest	Largest	Median	Average
free	563	860,805 KB	16 B	239,844 KB	80 B	1,528 KB
data object	30.317	922,953 KB	16 B	600 B	32 B	31 B
class object	2.260	371,289 KB	168 B	176 B	168 B	168 B
1-byte array (byte[], boolean[])	1.369	224,344 KB	24 B	2,148 KB	32 B	167 B
2-byte array (short[], char[])	11.010	632,234 KB	24 B	15,523 KB	48 B	58 B
4-byte array (object[], int[], float[])	2.658	223,773 KB	24 B	16,023 KB	40 B	86 B
8-byte array (long[], double[])	129	7,836 KB	32 B	2,000 KB	32 B	62 B
non-Java object	272	28,617 KB	16 B	8,023 KB	32 B	107 B

B+-Tree Serializado

Type	Count	Total Size	Smallest	Largest	Median	Average
free	994	544.180 KB	16 B	237.867 KB	128 B	560 B
data object	38,890	1.210 MB	16 B	552 B	32 B	32 B
class object	2,244	368.680 KB	168 B	176 B	168 B	168 B
1-byte array (byte[], boolean[])	1,369	224.289 KB	24 B	2.148 KB	32 B	167 B
2-byte array (short[], char[])	12,060	778.812 KB	24 B	15.523 KB	48 B	66 B
4-byte array (object[], int[], float[])	3,888	270.664 KB	24 B	16.023 KB	40 B	71 B
8-byte array (long[], double[])	128	7.625 KB	32 B	2.000 KB	32 B	61 B
non-Java object	325	30.625 KB	16 B	8.023 KB	32 B	96 B

Figura 72. Comparación de creación de tipos de datos

En la figura 72, se puede apreciar que en promedio se utiliza un 19% más de objetos que el Virtual Machinery, debido a que lo almacena en memoria, lo que se ve compensado con la

velocidad que imprime al árbol, al momento de realizar las operaciones y al tener un acceso rápido, los objetos no son tan distantes.

Con lo anterior se puede ver que el B+-Tree serializado crece en memoria con respecto al número de registros que éste almacena, por lo que veremos su comportamiento en diferentes escenarios:

Registros	Tamaño de Pila (Mb)	Memoria Alojada (Mb)	Memoria libre (Kb)	Número de Objectos
10	2.754	2.354	409.266	48431
100	2.82	2.381	449.648	49006
1000	3.383	2.757	640.492	57202
10000	4.508	3.703	824.289	77786
100000	4.57	3.836	751.586	80282
1000000	4.758	3.889	889.875	81875

Tabla 25. Análisis de expansión de memoria B+-Tree Serializado

En la tabla 25, se puede observar que a medida de que los registros incrementan, existe un aumento en el tamaño de la pila, de la memoria, por lo que se reconoce la limitación que la implementación del B+-Tree serializado es efectiva hasta los 300,000 registros, de ahí en más, ocupa demasiada memoria; para eso se pueden implementar varios mecanismos y seguir con la funcionalidad tales como:

- Diversificar el árbol en varios archivos a medida que éste vaya creciendo para ir leyendo sólo lo necesario e inyectar o eliminar en las ramas de forma rápida.
- Hacer un balanceo de carga entre el árbol y repartirlo en diversas variables para poder obtener el control dinámico del mismo.

Se presentan las capturas del desglose de la memoria con 1 millón de registros, podemos ver que el B+-Tree Serializado en la figura 73, incrementó su pila en un 31% con respecto a la del virtual Machinery en la figura 74, además de tener en memoria alojada de 38% para conservar el árbol, el número de objetos también incrementó en un 37%, en éste punto el árbol consume bastantes recursos del teléfono, por lo que se recomienda utilizar la implementación del B+-Tree con menos de 300,000 registros.

ID	Heap Size	Allocated	Free	% Used	# Objects
1	4.570 MB	3.836 MB	751.586 KB	83.94%	80,282

Cause GC

Display: Stats

Type	Count	Total Size	Smallest	Largest	Median	Average
free	1,692	721.211 KB	16 B	237.867 KB	128 B	436 B
data object	55,918	1.833 MB	16 B	744 B	32 B	34 B
class object	2,288	375.883 KB	168 B	176 B	168 B	168 B
1-byte array (byte[], boolean[])	1,372	224.375 KB	24 B	2.148 KB	32 B	167 B
2-byte array (short[], char[])	14,082	1.040 MB	24 B	15.523 KB	56 B	77 B
4-byte array (object[], int[], float[])	6,489	376.906 KB	24 B	16.023 KB	40 B	59 B
8-byte array (long[], double[])	133	8.258 KB	32 B	2.000 KB	32 B	63 B
non-Java object	325	30.625 KB	16 B	8.023 KB	32 B	96 B

Figura 73. B+-Tree 1,000,000 registro en memoria

ID	Heap Size	Allocated	Free	% Used	# Objects
1	3,195 MB	2,404 MB	810,258 KB	75,24%	50.840

Cause GC

Display: Stats

Type	Count	Total Size	Smallest	Largest	Median	Average
free	644	781,891 KB	16 B	239,844 KB	80 B	1,214 KB
data object	32.171	963,016 KB	16 B	600 B	32 B	30 B
class object	2.265	372,117 KB	168 B	176 B	168 B	168 B
1-byte array (byte[], boolean[])	2.574	258,094 KB	24 B	2,148 KB	32 B	102 B
2-byte array (short[], char[])	11.010	632,688 KB	24 B	15,523 KB	48 B	58 B
4-byte array (object[], int[], float[])	2.692	227,680 KB	24 B	16,023 KB	40 B	86 B
8-byte array (long[], double[])	128	7,750 KB	32 B	2,000 KB	32 B	62 B
non-Java object	272	28,617 KB	16 B	8,023 KB	32 B	107 B

Figura 74. Virtual Machinery 1,000,000 registros

Resumiendo, el B+-Tree al tener conjuntos de datos grandes tiene una debilidad, debido a que el árbol se mantiene en memoria y consume muchos recursos, por lo que deben de optimizar su funcionamiento cuando trabaje con la memoria.

6.4 Tamaño de los archivos

Se presenta la siguiente tabla que revela el tamaño de los archivos generados por las diferentes implementaciones, las cantidades están expresadas en kb, cabe hacer notar que por parte de los nodos serializados no se pudieron obtener datos después mayores o iguales a 1000 registros, debido al error comentado en el apartado 6.2.

Número de registros	Virtual Machinery	B+-Tree Serializado	B+-Tree Acceso Aleatorio	B+-Tree Nodos Serializados
100	16 Kb	2 Kb	8.7 Kb	18 Kb
1000	64 Kb	19 Kb	90 Kb	No disponible

10000	536 Kb	182 Kb	878 Kb	No disponible
100000	5284 Kb	1856 Kb	8790 Kb	No disponible

Tabla 26. Comparación de tamaños en disco

Se puede presumir, que en espacio en disco, el B+-tree Serializado, barre a los demás, debido al algoritmo de compresión de objetos que tiene implícito Java, así puede almacenar objetos enteros que conservan sus propiedades, enlaces y objetos internos. Esta prueba fue la más satisfactoria de todas al poder derrotar en espacio en disco al Virtual Machinery en un 72%. Demostrando que ocupa menos espacio en la memoria flash del dispositivo, la gráfica comparativa puede apreciarse en la figura 75.

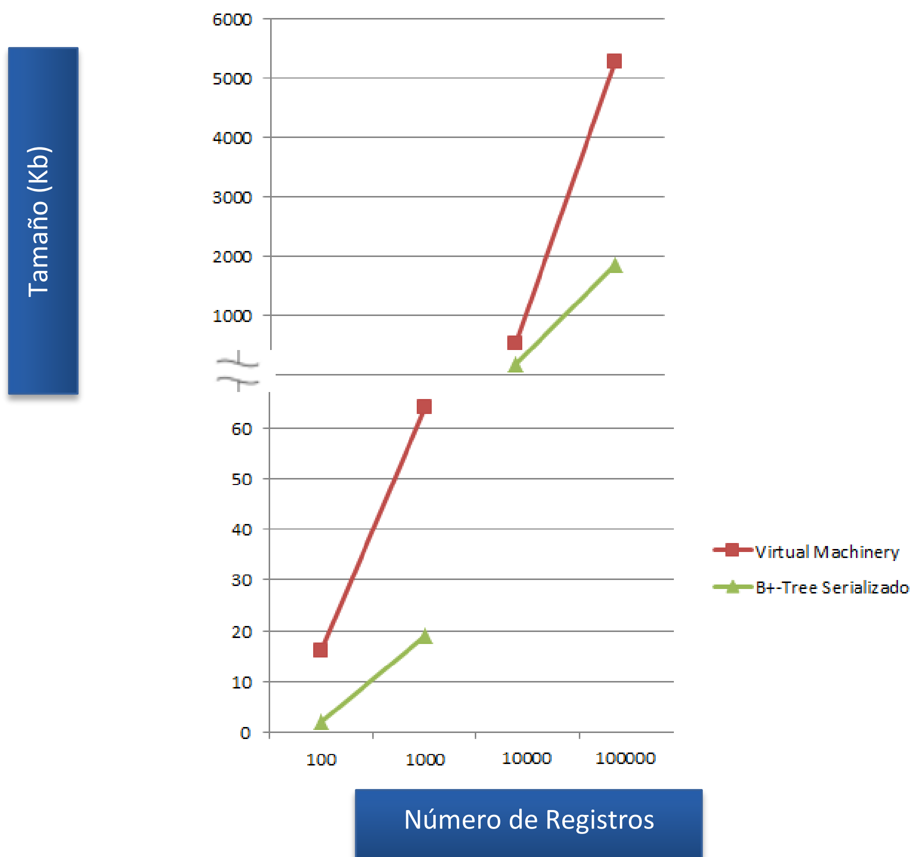


Figura 75. Comparación de tamaño entre implementaciones

7. Capítulo Séptimo: Conclusiones y trabajo futuro

Al implementar un B+-Tree o cualquier algoritmo en un dispositivo móvil con sistema operativo Android, es necesario de tomar en cuenta las limitaciones del mismo, ya que no podemos darnos el lujo de derrochar memoria RAM, costo de procesamiento, consideraciones energéticas y/o espacio en disco. A lo largo del trabajo se analizaron diversas maneras de contrarrestar las limitaciones de un dispositivo móvil, implementándose las principales para obtener una aplicación que mantuviera un rendimiento óptimo en cualquier gadget con Android.

El resultado de éste trabajo de investigación es un prototipo del algoritmo B+-Tree en Android, tomando a consideración las limitaciones del dispositivo y resultando ser funcional en conjuntos de datos moderados. Otros resultados son:

- En el capítulo 5 se ilustró su uso mediante un ejemplo básico para desarrollar cualquier aplicación que requiere indexación.
- Consejos para realizar una aplicación para Android atendiendo a las necesidades de memoria, espacio y procesamiento.
- Comparación entre varios métodos de persistencia.
- Benchmarking sobre varias derivaciones del B+-Tree
- Análisis de la arquitectura Android y su implicación en el desarrollo eficiente de aplicaciones.
- Reseña histórica del B+-Tree.

Así mismo, el trabajo revela que la mejor implementación (B+-Tree Serializado) funciona de manera óptima con un conjunto de datos pequeños, menores a 300,000 registros, superando en el tiempo de respuesta al algoritmo comercial del Virtual Machinery. Esto nos sugiere varios puntos para tratarlos como trabajo futuro:

- Optimizar el algoritmo del B+-Tree Serializado.
- Manejar hilos para poder hacer inserciones/eliminaciones/consultas de manera paralela, utilizando el uso de transacciones para no desquebrajar la estructura del árbol.
- Implementar un uso inteligente de la serialización, es decir, cuando se tenga un árbol de tamaño considerable se podría serializar por partes, permitiendo cargar en memoria sólo las ramas necesarias para ejecutar las operaciones deseadas.
- Mejorar la clase Nodo para que pueda resolver cualquier tipo de dato primitivo o bien un objeto entero.
- Diseñar e implementar un mecanismo óptimo para visualizar y/o interactuar con el árbol. Por ejemplo: Una estructura que permita por medio de la pantalla táctil el

insertar/eliminar/consultar directamente en el árbol, sin la necesidad de entrar a formularios.

- Hacer mejoras sobre el algoritmo B+-Tree: Una pequeña mejora podría ser el implementar en vez de una lista doble, una lista circular; otra mejora podría ser el personalizar el número de nodos que pudiese contener cada nivel del árbol B+-tree.
- Tomar en cuenta las propiedades del dispositivo de almacenamiento y no sólo las del ambiente Android. Una memoria flash no soporta las operaciones requeridas por una estructura de datos a la manera tradicional de un disco magnético por lo que se tienen penalizaciones del performance en operaciones que impliquen actualización.
- Completar y corroborar los resultados obtenidos con el emulador, usando un dispositivo real.

Finalmente se considera que el B+-tree a pesar de ser un algoritmo tan viejo, podrá seguir siendo útil para muchas implementaciones futuras.

Referencias

[Bayer 1972] Bayer, R. and McCreight, E.1972. Organization and Mantenience of large Ordered Indexes, Acta informática, Vol 1, Fasc 3,Computer Science Department, Purdue University.

[Bayer 1977] Bayer, R. and Unterauer, K.1977. Prefix B-Trees,Technische Universität München.

[Becker 1993] Becker, B., Gschwind, S., Ohler, T., Seege,r B., Widmeyer, P.1993. An Asymptotically Optimal Multiversion B-Tree, Philipps-Universität.

[Blanco 2007] Blanco, J. 2007-2008. Notas de Almacenamiento y Recuperación de la información Central de Venezuela, Facultad de Ciencias, Escuela de Computación.

[Brodal 2012] Brodal, G., Sioutas, S., Tsakalidis, K. and Tschilas, K. 2012. Fully Persistent B-Trees, Department of computer Science Aarus University, Denmark.

[Chen 2010] Chen, H, Qiang, L. and Peiquan, J. 2010. A new index for Temporal Information in Web Pages, School of Computer Science and Technology

[Comer 1979] Comer, D., 1979. The Ubiquitous B-Tree, Computer Science Department, Purdue University.

[Dinadayalan 2011] Dinadayalan, D., Dinadayalan, G. 2011. Interpolation B-Tree: A New Access Method, Department of Computer Science Kanchi Mamunivar centre for P.G. Studies Pondicherry, India.

[Dongui 2003] Donghui, Z., 2003. B-Trees,Northeastern University.

[Ehringer 2010] Ehringer, D., 2010. The Dalvik Virtual Machine Architecture March.

[Gap-Joo 2011] GAP-JOO , N.A., BONGKI, M. AND LEE, S., 2011, IPL B+-tree for Flash Memory Database Systems, Sungkyunkwan University.

[Graefe 2011] Graefe, G., 2011, Modern B-tree techniques, Packard Laboratories.

[Guihot 2012] Guihot, H., 2012. Pro Android Apps Performance Optimization, Apress.

[Guttman 1984] Guttman, A., 1984. R-Trees a Dynamic Index Structure for Spacial Searching University of California Berkley.

[Hashimi 2010] Hashimi, S., Komatineni, S. and MacLean, D., 2010 . Covers Google's Android 2 Plataform including advanced topics such as OpenGL, Widgets, Text to Speech, Multi-Touch, and Titanium Mobile.

[IBM 2009] IBM, S.G., 2009. Websphere Application Server V6.1 Problem Determination Guide, IBM.

[Jensen 2004] Jensen, C.S., Lin, D., Chin, B., 2004. Query and Update Efficient B+-Tree Based indexing of Moving Objects, Departament of Computer Science Aalborg University, Denmark.

[Joyanes 2006] Joyanes, L., 2006. C++ Algoritmos, estructuras de datos y objetos, S.E. España.

[Kaltenbrunner 2009] Kaltenbrunner, A., Kellis, L. AND Mart, D., 2009. B-Trees.

[Knuth 1973] Knuth, 1973. The art of computer programming Vol 3: Sorting and searching Addison-wesley.

[Kumar 2008] Kumar, A., 2008. First Look At Android January.

[Lanka 2011] Lanka, S. and Mays, E., 2011. Fully Persistent B+-Trees, Computer Science Department The Pennsylvania state University, IBT T.J. Watson Research Center.

[Lee] Lee, A.H. and Shin, H.Building a Persistent Object Store using the Java Reflection API Programming systems Laboratory, Korea University.

[Luo 2011] Luo, G., 2011. Locking Protocol for Materialized Aggregate Join Views on B-tree Indices, IBM T.J. Watson Research Center.

[Manolopoulos 2010] Manolopoulos, Y., 2010, R-trees Have Grown Everywhere, University of Piraeus Greece.

[Nascimento 1998] Nascimento M.A. and Dunham, M.A., 1998. Indexing Valid Time Databases Via B+-Tree The Map21 Approach, Southern Methodist University.

[Palakodety 2010] Palakodety, R. Faisal, M. and Lin, W., 2010. Building and Configuring a Real-Time indexing System, Oracle.

[Tadepalli 2006] Tadepalli, P., 2006. Grid-Based Distributed Search Structure, University of Mississippi.

[Tao 2010] Tao, Y., 2010.The B-tree and its Variants (Part II - Persistent B-tree),Chinese University of Hong Kong.

[Tuch 2009] Tuch, H., Laplace, C., Barr,K.C. and Wu, B., 2009. Block Storage Virtualization With Commodity Secure Digital Cards.

[Twigg 2010] Twigg, A., Byde A, Milo, G., Moreton, T.,Wilkesy, J. and Wilkie, T., 2010, Stratified B-trees and Versioned Dictionaries, Google.

[Wu 2008] WU, C. and KUO, T.,2008. An efficient B-Tree layer implementation for flash memory Storage Systems, National Taiwan University, National Chiao-Tung University.

[Wu 2010] Wu, S., Jiang, D., Chin, B. and Wu, K., 2010. Efficient Btree Based Indexing for Cloud Data Processing, School of Computing, National University of Singapore.

[Xiang 2006] Xiang, X, Yue, L., Liu, Z. and Wei, P., 2006. A Reliable B-Tree Implementation over Flash Memory, University of Science and Technology of China Hefei, P.R.China.

Referencias Web

[VirtualMachinery 2012] Virtual Machinery, 2012. B+-Tree Implementation.
http://www.virtualmachinery.com/btreeprod.htm

[Mysql 2012] Mysql, 2012. InnoDb Data Storage and Compression.
http://dev.mysql.com/doc/innodb/1.1/en/innodb-compression-internals.html

[RBTOracle 2012] Oracle, 2012. Red Black Tree.
http://docs.oracle.com/javase/1.5.0/docs/api/java/util/TreeMap.html

[SplayTree 2012] MagickCore, 2012. Splay-Tree Reference.
http://www.imagemagick.org/api/MagickCore/splay-tree_8c.html

[AndroidConferences 2012] Android, 2012. The Dalvik Virtual Machine
http://www.youtube.com/watch?v=ptjedOZEXPM

[Herogyang 2012] Herogyang, 2012.
Introduction of Activity Lifecycle
http://www.herongyang.com/Android/Activity-Introduction-of-Activity-Lifecycle.html

[OracleEntityPersistent 2012] Oracle, 2012. The Java Persistence API
http://www.oracle.com/technetwork/articles/Javaee/jpa-137156.html

[OracleSerialization 2012] Oracle, 2012.
Java Object Serialization Specification
http://docs.oracle.com/javase/7/docs/platform/serialization/spec/serialTOC.html

ANEXOS