UNIVERSIDAD NACIONAL JORGE BASADRE GROHMANN - TACNA

Escuela de Posgrado

MAESTRÍA EN INGENIERÍA DE SISTEMAS E INFORMÁTICA
ADMINISTRACIÓN DE TECNOLOGÍAS DE INFORMACIÓN

VELOCIDAD DE RESPUESTA EN LA BÚSQUEDA DE DATOS ALMACENADOS EN ESTRUCTURAS DE DATOS DINÁMICAS

TESIS

PRESENTADA POR:

ING. HUGO MANUEL BARRAZA VIZCARRA

Para optar el Grado Académico de:

MAESTRO EN CIENCIAS (*MAGISTER SCIENTIAE*) CON MENCIÓN EN INGENIERÍA DE SISTEMAS E INFORMÁTICA – ADMINISTRACIÓN DE TECNOLOGÍAS DE INFORMACIÓN

TACNA - PERÚ

2017

UNIVERSIDAD NACIONAL JORGE BASADRE GROHMANN

Escuela de Posgrado

MAESTRÍA EN INGENIERÍA DE SISTEMAS E INFORMÁTICA
ADMINISTRACIÓN DE TECNOLOGÍAS DE INFORMACIÓN

VELOCIDAD DE RESPUESTA EN LA BÚSQUEDA DE DATOS ALMACENADOS EN ESTRUCTURAS DE DATOS DINÁMICAS

Tesis sustentada y aprobada el 11 de julio del 2017; estando el jurado calificador integrado por:

PRESIDENTE : ....
Dra. Karin Yanet Supo Gavancho

SECRETARIO : ....
Mgr. Gianfranco Alexey Málaga Tejada

MIEMBRO : ....
M.Sc. Edgar Aurelio Taya Acosta

ASESOR : ....
Dr. Edwin Antonio Hinojosa Ramos

Agradecimiento

A lo largo de este proceso siempre hubo gente que en todo momento me impulsaba en seguir adelante. Es por eso que quisiera dar mi gratitud a todas esas personas especiales que siempre me brindaron su apoyo, confianza, amistad y cariño.

Quiero agradecer con mucho amor y cariño a mis padres, que me apoyaron en toda esta travesía, tanto en mi vida personal como en mi vida universitaria, estando conmigo en los momentos buenos y en los malos.

Agradezco igualmente a mis maestros por su generosa contribución al compartir sus conocimientos y sabiduría.

Y por supuesto, a mi gran Dios.

Dedicatoria

A Dios, ese Ser Todopoderoso que nos dio la vida y que siempre nos acompaña, a mis queridos padres Hugo y María, por todo el sacrificio para hacer de mí una persona de bien.

CONTENIDO

Pág

Agradecimiento iii

Dedicatoria iv

RESUMEN x

ABSTRACT xi

INTRODUCCIÓN 1

CAPÍTULO I: PLANTEAMIENTO DEL PROBLEMA 3

1.1. Descripción del problema 3

1.1.1. Antecedentes del problema 3

1.1.2. Problemática de la investigación 4

1.2. Formulación del problema 5

1.3. Justificación e importancia de la investigación 5

1.4. Alcances y limitaciones 6

1.5. Objetivos 7

1.5.1. Objetivo general 7

1.5.2. Objetivos específicos 7

1.6. Hipótesis 7

1.6.1. Hipótesis general 7

1.6.2. Hipótesis específica 8

CAPÍTULO II: MARCO TEÓRICO 9

2.1. Antecedentes del estudio 9

2.2. Bases teóricas 11

2.2.1. Estructuras de datos dinámicas 11

A. Árboles Binarios 12

B. Árboles Binarios de Búsqueda 19

C. Árboles Binarios de Búsqueda Auto-balanceados 21

2.2.2. Búsqueda de datos 44

2.3. Definición de términos 49

CAPÍTULO III: MARCO METODOLÓGICO 51

3.1. Tipo y diseño de la investigación 51

3.2. Población y muestra 51

3.3. Operacionalización de variables 52

3.4. Técnicas e instrumentos para recolección de datos 53

3.5. Procesamiento y análisis de datos 54

CAPÍTULO IV: RESULTADOS 60

CAPÍTULO V: DISCUSIÓN 67

CONCLUSIONES 74

RECOMENDACIONES 76

REFERENCIAS BIBLIOGRÁFICAS 77

ANEXOS 80

ÍNDICE DE TABLAS

Pág

Tabla 1 Árboles Fibonacci, hasta h=4 28

Tabla 2 Secuencia números de Fibonacci 30

Tabla 3 Códigos de las especialidades de la UNJBG 55

ÍNDICE DE FIGURAS

		Pág
Figura 1	Estructura tipo árbol.	14
Figura 2	Representación de un árbol binario.	17
Figura 3	Árbol binario de búsqueda	21
Figura 4	Número máximo de nodos en un árbol binario completo.	22
Figura 5	ABB con crecimiento descontrolado.	24
Figura 6	Ejemplos de Árboles AVL	26
Figura 7	Árbol Fibonacci para h=5	29
Figura 8	Árbol Red–Black.	35
Figura 9	Complejidad de las alturas de árboles.	40
Figura 10	Operaciones en un árbol splay.	43
Figura 11	Diagrama de componentes de la aplicación desarrollada	57
Figura 12	Diagrama de clases de la aplicación desarrollada	58
Figura 13	Menú del programa utilizado para generar los registros	59
Figura 14	Archivo de texto CSV con 6 966 registros	60
Figura 15	No redundancia de datos del campo código	61
Figura 16	Datos correspondientes a tres muestras de 99 registros para el número de comparaciones en la búsqueda de datos.	62

Figura 17 Estadísticas descriptivas para las comparaciones en las 03 muestras 62
Figura 18 Prueba T para los tiempos de respuesta en la muestra 1 68
Figura 19 Prueba T para los tiempos de respuesta en la muestra 2 69
Figura 20 Prueba T para los tiempos de respuesta en la muestra 3 71
Figura 21 Prueba T para las comparaciones 72

RESUMEN

La presente investigación tiene por objetivo estudiar la velocidad de respuesta en la búsqueda de datos almacenados en estructuras de datos dinámicas, específicamente estructuras del tipo árbol binario de búsqueda auto-balanceado. La principal ventaja de almacenar datos en estructuras dinámicas es obtener mayor velocidad de respuesta en las operaciones de búsqueda de datos específicos en comparación con las estructuras estáticas o las estructuras lineales. En el presente trabajo se comparó dos estructuras dinámicas y se observó que el tiempo que tardan en insertarse los datos difiere dependiendo de la estructura elegida y del orden en cómo son ingresados los datos.

Palabras clave: Estructura de datos, árbol binario de búsqueda, velocidad de respuesta, búsqueda de datos.

ABSTRACT

The present research aims to study the speed of response in the search of data stored in dynamic data structures, specifically structures of the self-balanced binary tree type. The main advantage of storing data in dynamic structures is to obtain the highest response speed in specific data search operations as compared to static structures or linear structures. In the present work we compared the dynamic structures of the self-balanced binary tree type and observed that the time took it to insert the data differs depending on the structure chosen and the order in which the data are entered.

Keywords: Data structure, binary search tree, response speed, data search.

INTRODUCCIÓN

En todo sistema de computación es necesario almacenar datos con el fin de utilizarlos, en un futuro, de manera eficiente. Estos datos se pueden almacenar en la memoria principal o secundaria del computador. La forma en cómo se organizan estos datos se conoce como estructura de datos. Existen distintas formas de organización, y cada una posee diferentes características que hacen que sean más o menos ventajosas para diferentes procesos en el ordenador. Por lo tanto, el objetivo fundamental de toda estructura de datos es organizar los datos en la memoria del computador para poder utilizarlos y operarlos en la ejecución de los programas de manera eficiente.

En Ciencias de la Computación, las estructuras de datos se clasifican en dos grandes grupos: estructuras estáticas y estructuras dinámicas. Las primeras no pueden variar su tamaño ni su capacidad de almacenamiento durante la ejecución del programa, lo que representa un riesgo de desbordamiento cuando la cantidad de datos supera a su capacidad de almacenamiento. En cambio las estructuras dinámicas pueden modificar su tamaño y capacidad de almacenamiento en plena ejecución del programa, con lo cual se consume sólo el espacio de memoria necesario. Con esto se elimina el riesgo de desbordamiento de la

estructura. Pero es importante indicar que la complejidad del diseño y de los algoritmos que trabajen sobre datos contenidos en estructuras dinámicas es más alta con respecto a los algoritmos que trabajen sobre datos contenidos en estructuras estáticas.

En el presente trabajo de investigación se estudian dos estructuras dinámicas, uno de ellas es el árbol AVL y la otra es el árbol *Red–Black*. Se estudia el tiempo de inserción de datos y el tiempo de respuesta para los procedimientos de búsqueda de datos y con ello determinar la velocidad de respuesta de los algoritmos de búsqueda que actúan sobre estas estructuras. Para ello se implementó, en el lenguaje de programación C++, una aplicación que extrae datos de la memoria secundaria y los inserta en las dos estructuras y a partir de ello efectuar operaciones de búsqueda, cronometrando el tiempo de respuesta y contando el número de comparaciones de los procedimientos, y comparar sus desempeños.

CAPÍTULO I

PLANTEAMIENTO DEL PROBLEMA

1.1. Descripción del problema

1.1.1. Antecedentes del problema

Según Loomis (1999, p. 3), el almacenamiento de datos es un proceso costoso. Es por esto que deben ser manejados de tal manera que se mantenga su integridad y que estén disponibles para producir información.

Para que los datos sean manipulados correctamente y produzcan información, es necesario de algoritmos que los gestionen. Muchos de estos algoritmos requieren una forma de representación para lograr ser eficientes. Esta forma de representación de los datos junto con las operaciones permitidas sobre estos datos se conoce como estructura de datos. Todas las estructuras de datos permiten inserciones sin una condición específica. Las estructuras de datos varían en cómo permiten el acceso a miembros del grupo. Algunas permiten tanto accesos como operaciones de borrado arbitrarios. Otras imponen restricciones, tales como permitir el acceso sólo al elemento más recientemente insertado. (Weiss, 2004, p. 137).

Murillo Morera, J. (2012) en su artículo científico titulado: Comparación entre algoritmos recursivos e iterativos y su medición en términos de eficiencia realiza comparaciones simples entre algoritmos recursivos e iterativos para determinar sus grados de eficiencia ante un problema en particular. Se efectuaron pruebas de comparación y análisis utilizando tres ejemplos en ambos tipos de algoritmos, a los cuales se les aplicaron los criterios de análisis de algoritmos. El autor concluye que los procesos recursivos sólo se deben emplear en casos que no se puedan resolver por métodos iterativos.

1.1.2. Problemática de la investigación

Actualmente estamos en una época donde la información ha tomado un rol primordial en todas las áreas del conocimiento y en toda actividad humana. Para almacenar dicha información en equipos informáticos se requiere de una gran capacidad de almacenamiento y de equipos con recursos de mayor potencia para poder reducir el tiempo de acceso y recuperación de datos.

Además, también es importante la forma lógica con la que se organiza y se estructura dicha información; esto, en Ciencias de Computación, se conoce como estructura de datos. Las estructuras de

datos se clasifican en dos grandes grupos: estructuras estáticas y estructuras dinámicas, siendo las más utilizadas, las estructuras dinámicas.

La información se almacena en el computador con el fin de ser recuperada, en un futuro, para realizar operaciones sobre ella. La operación más común en computación, es la búsqueda de datos. Si los datos han sido estructurados en la memoria principal bajo un modelo lógico inadecuado, entonces no se alcanzarán velocidades de respuesta eficientes en la búsqueda de datos.

1.2. Formulación del problema

Ante esto, surge la siguiente interrogante: ¿Cómo influyen las estructuras de datos dinámicas en la velocidad de respuesta de la búsqueda de datos en la memoria principal?

1.3. Justificación e importancia de la investigación

Todo sistema informático se basa en la información que este gestiona. Esta información se almacena generalmente en dispositivos externos y, para su uso posterior, debe extraerse y estructurarse en la memoria del computador y es ahí donde se realizan operaciones sobre ella.

Como se dijo anteriormente, las estructuras que se utilicen para contener los datos en la memoria son importantes para poder alcanzar diferentes velocidades de localización de los datos buscados. Por ello es importante estudiar cómo funcionan los algoritmos de búsqueda de datos de estructuras de datos dinámicas y comparar cuál de estas estructuras genera mejores velocidades de respuesta.

1.4. Alcances y limitaciones

En el presente trabajo de investigación se realiza un estudio sobre los tiempos de respuesta en la búsqueda de los registros de las matrículas realizadas en el periodo académico 2017 – I de la Universidad Nacional Jorge Basadre Grohmann. Estos registros estarán almacenados en la memoria secundaria para luego ser insertados en estructuras de datos dinámicas residentes en la memoria principal del computador.

El estudio se centra en el uso de dos estructuras dinámicas de tipo árbol binario de búsqueda auto-balanceado: los árboles AVL y los árboles *Red–Black*.

1.5. Objetivos

1.5.1. Objetivo general

Evaluar la velocidad de respuesta en la búsqueda de datos almacenados en estructuras de datos dinámicas.

1.5.2. Objetivos específicos

- Determinar los tiempos de respuesta en la búsqueda de datos almacenados en estructuras de datos dinámicas.
- Determinar el número de comparaciones en la búsqueda de datos almacenados en estructuras de datos dinámicas.

1.6. Hipótesis

1.6.1. Hipótesis general

H_0: Existe diferencia significativa entre las velocidades de respuesta para localizar un dato en una estructura dinámica del tipo árbol AVL y en una estructura dinámica del tipo *Red–Black*.

H_a: No existe diferencia significativa entre las velocidades de respuesta para localizar un dato en una estructura dinámica del tipo árbol AVL y en una estructura dinámica del tipo *Red–Black*.

1.6.2. Hipótesis específicas

Hipótesis específica 1:

H_0: Existe diferencia significativa entre los tiempos de respuesta para localizar un dato en una estructura dinámica del tipo árbol AVL y en una estructura dinámica del tipo *Red–Black*.

H_1: No existe diferencia significativa entre los tiempos de respuesta para localizar un dato en una estructura dinámica del tipo árbol AVL y en una estructura dinámica del tipo *Red–Black*.

Hipótesis específica 2:

H_0: Existe diferencia significativa entre el número de comparaciones necesarias para localizar un dato en una estructura dinámica del tipo árbol AVL y en una estructura dinámica del tipo *Red–Black*.

H_1: No existe diferencia significativa entre el número de comparaciones necesarias para localizar un dato en una estructura dinámica del tipo árbol AVL y en una estructura dinámica del tipo *Red–Black*.

CAPÍTULO II

MARCO TEÓRICO

2.1. Antecedentes del estudio

Andersson (1993) en su artículo científico “*Balanced Search Tree Made Simple*” indica que existe una enorme brecha entre la computación teórica y la computación aplicada, y pone como ejemplo un caso específico: los algoritmos eficientes para búsqueda que se imparten en los cursos introductorios, son a menudo reemplazados por métodos pobres, como la ordenación por el método de la burbuja y el uso de listas enlazadas. Si el programa resulta requerir demasiado tiempo, la solución aconsejada es comprar una computadora nueva, más actualizada y más rápida.

Pfaff (2004) en su artículo científico titulado “*Performance Analysis of BSTs in System Software*” indica que los árboles binarios de búsqueda son estructuras que se usan a menudo para la estructuración de la memoria principal así como en núcleos de sistemas operativos; elegir el tipo de árbol adecuado puede afectar significativamente el rendimiento, pero la literatura ofrece pocos estudios empíricos sobre esto. Pfaff realiza una comparación entre distintas variantes de árboles binarios de búsqueda en distintos

escenarios y concluye que las operaciones sobre estos dependen de la forma en cómo serán insertados los datos en los árboles.

Heger (2004) en su artículo científico titulado “*A Disquisition on The Performance Behaviour of Binary Search Tree Data Structures*”, nos da una idea de las situaciones en que se deben usar los diferentes tipos de árboles binarios de búsqueda más usados.

Si las claves siempre ingresan en orden aleatorio, puede emplearse un árbol binario de búsqueda, sin embargo, si por momentos se producen ingresos de claves en orden creciente o decreciente, debe escogerse una estructura que pueda rebalancear dinámicamente el árbol ya que un árbol binario de búsqueda presentaría un crecimiento descontrolado.

Bronson, Casper, Chafi y Olukotun (2010) en su artículo científico titulado “*A practical concurrent binary search tree*” proponen un algoritmo binario de búsqueda concurrente que reduce los gastos generales y que evita rebalanceos innecesarios. Los diversos experimentos que realizaron demuestran que su algoritmo supera a la versión secuencial del mismo.

Hinojosa (2014) en su tesis titulada “Velocidad de respuesta de los algoritmos de búsqueda de datos contenidos en estructuras estáticas y dinámicas” realiza una comparación entre una estructura estática del tipo arreglo unidimensional y una estructura dinámica del tipo árbol auto-

balanceado en función a los tiempos de respuesta y el número de comparaciones en la búsqueda de datos de 1 583 registros de postulantes a la UNJBG. El autor concluye que la velocidad de respuesta en la búsqueda de datos está en función del tipo de estructura de datos que se utilice para contener los datos.

2.2. Bases teóricas

2.2.1. Estructuras de datos dinámicas

Una de las aplicaciones más interesantes y potentes del uso de la memoria son las estructuras dinámicas de datos. Las estructuras de datos estáticas tienen una importante limitación: no pueden cambiar de tamaño durante la ejecución (S.I. Didact, 2005).

En muchas ocasiones se necesitan estructuras que puedan cambiar de tamaño durante la ejecución del programa. Esto se consigue con las estructuras dinámicas. Las estructuras dinámicas usan solamente la cantidad de memoria que necesitan en cada momento. Si una estructura dinámica contiene 10 elementos, reservará espacio en memoria para 10. Pero si de repente pasa a contener 5 elementos, usará espacio en memoria para 5. Esto garantiza un uso muy efectivo de la memoria.

Las estructuras dinámicas teóricamente no tienen límite de almacenamiento, pero en la práctica su límite de almacenamiento vendrá dado únicamente por la cantidad de memoria del ordenador.

Además, las estructuras dinámicas permiten crear estructuras de datos muy flexibles, ya sea en cuanto al orden, la estructura interna o las relaciones entre los elementos que las componen.

A. Árboles Binarios

Los árboles son las estructuras de datos dinámicas y no-lineales más importantes en la computación. Son dinámicas, puesto que la estructura de tipo árbol puede cambiar en tiempo de ejecución del programa. Y son no-lineales, puesto que a cada elemento del árbol pueden seguirle uno o varios elementos.

A continuación se citan algunas de las definiciones dadas por diferentes autores entendidos en el tema:

Intuitivamente el concepto de árbol implica una estructura en la que los datos se organizan de modo que los elementos de información están relacionados entre sí a través de ramas. Un árbol consta de un conjunto finito de elementos, denominados «nodos» y un conjunto finito de líneas

dirigidas, denominadas «ramas» que conectan los nodos (Joyanes, 2002, p. 499).

La organización de los datos en una estructura no lineal, jerárquica o de niveles, es una opción para representar estructura de datos; las más conocidas y usadas en la computación se denominan árboles. Su característica principal es que mantienen una relación de uno a muchos (1:n) entre sus elementos (Martinez & Quiroga, 2002, p. 116).

"Formalmente se define un árbol de tipo T como una estructura homogénea que es la concatenación de un elemento de tipo T junto con un número finito de árboles disjuntos, llamados subárboles. Una forma particular de árbol puede ser la estructura vacía" (Cairó & Guardati, 2002, p. 170).

En base a las definiciones anteriormente mencionadas, se puede decir que un árbol, en computación, es una estructura de datos no lineal, jerárquica y dinámica compuesta por un conjunto de objetos denominados nodos (el nodo que se ubica en la parte superior del árbol de denomina nodo raíz).

Los árboles son estructuras muy comunes, se pueden encontrar varios ejemplos en la vida cotidiana (Bowman, 1999). La mayoría de las personas, por ejemplo, denominan *árbol genealógico* a su linaje.

La estructura tipo árbol se expresa generalmente de manera gráfica como se muestra en la figura 1.

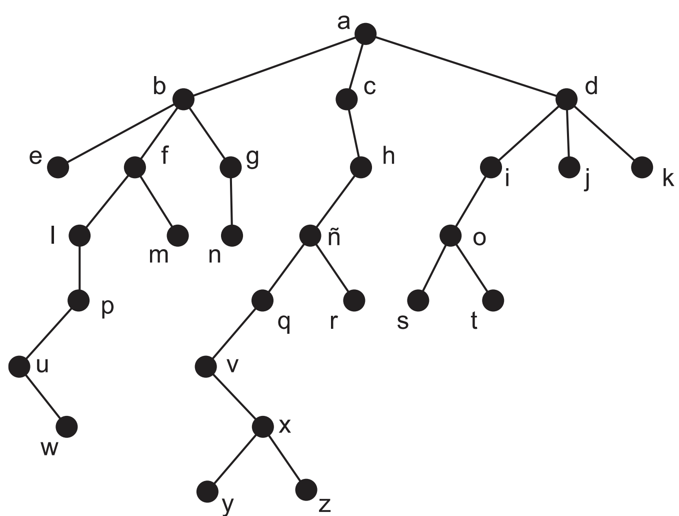


Figura 1. Estructura tipo árbol.
Fuente: Elaboración propia

La naturaleza jerárquica que tiene la organización de los nodos genera que existan relaciones de parentesco entre los nodos, dando lugar a términos como padre, hijo, antecesor, sucesor, etc.

El nodo *a* que se encuentra en la parte superior se denomina nodo raíz y los nodos *e*, *w*, *m*, *n*, *y*, *z*, *r*, *s*, *t*, *j* y *k* que no tienen descendientes se denominan nodos hoja. El resto de nodos se denominan nodos interiores.

En Ciencias de la Computación, “los árboles se usan exhaustivamente como una estructura eficiente para realizar búsquedas en conjuntos de datos grandes y para aplicaciones diversas como los núcleos

de los sistemas operativos, los sistemas de inteligencia artificial y los algoritmos de codificación”. (Forouzan & Chung, 2003, p. 334)

Definición de la clase Árbol Binario

Un árbol binario es un árbol en el que el número máximo de descendientes (o nodos hijo) que puede tener cualquiera de sus nodos es 2. Estos árboles son de especial interés puesto que representan una de las estructuras de datos más usadas en computación.

Un árbol binario es un conjunto finito de elementos que puede estar vacío o contener un elemento denominado la «raíz del árbol» y otros elementos divididos en dos subconjuntos separados, cada uno de los cuales es en sí un árbol binario. Estos dos subconjuntos son denominados subárbol izquierdo y subárbol derecho del árbol original. Cada elemento de un árbol binario se denomina un nodo del árbol. (Tanenbaum & Augenstein, 1993, p. 329)

Representación de un Árbol Binario en memoria

Cuando se necesita implementar un árbol binario en una computadora mediante un lenguaje de programación, se tiene que expresar el árbol en una forma que comprenda el computador (Cairó & Guardati, 2002, p. 180).

Tradicionalmente se usan dos formas.

- Por medio de arreglos.
- Por medio de enlaces tipo puntero.

Para implementar un árbol como un arreglo, un nodo se declara como un objeto con un campo de información y dos campos de «referencia». Estos campos de referencia contienen los índices de las celdas del arreglo en el cual se almacenan los hijos izquierdo y derecho si existen. Sin embargo, “esta implementación puede ser poco conveniente. Las localidades de los hijos deben conocerse para insertar un nodo nuevo. Después de eliminar un nodo del árbol, tendría que eliminarse un hueco en el arreglo”. (Drozdek, 2007, p. 219)

De lo anterior se desprende que implementar un árbol binario mediante un arreglo es ineficiente debido al costo que implica insertar o eliminar un elemento, es por esto que en el presente trabajo se estudiará la segunda forma de representar un árbol binario, es decir, por medio de enlaces tipo puntero, conocidos también como apuntadores.

Los nodos del árbol binario se representan como registros, tal como se ilustra en la figura 2, y cada uno de ellos contendrá tres campos.

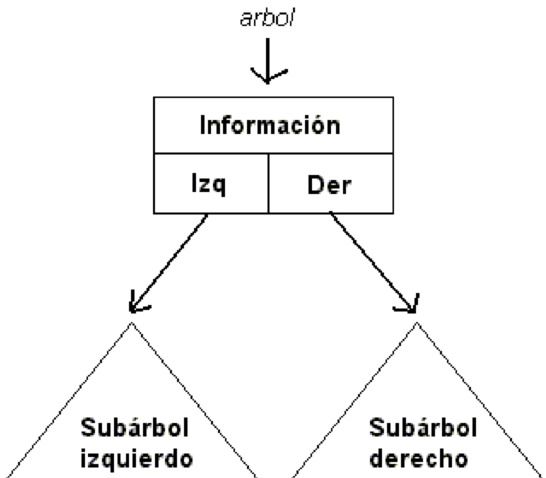


Figura 2. Representación de un árbol binario.
Fuente: Elaboración propia

El primer campo almacena la información del nodo. Los dos campos restantes se utilizan para apuntar el subárbol izquierdo y el subárbol derecho del nodo en cuestión.

Información: Campo donde se almacena la información de interés del nodo. Con fines didácticos, se suele almacenar en este campo un valor numérico. Pero se debe tener en cuenta que en la práctica se almacena en este campo registros, objetos e inclusive arreglos.

Izq*:* Campo donde se almacena la dirección de memoria del subárbol izquierdo del árbol.

Der: Campo donde se almacena la dirección de memoria del subárbol derecho del árbol.

Ha de notarse que esta representación de un árbol binario tiene una naturaleza recursiva debido a que el árbol binario contiene dos subárboles, que su vez son árboles binarios. Esta naturaleza facilita algunas operaciones que se realizan sobre el árbol binario, como por ejemplo, su recorrido, la inserción y eliminación de nodos, la búsqueda de una clave, entre otras operaciones.

Maneras de recorrer un árbol binario

Los programas que trabajan con árboles necesitan visitar todos sus nodos para conocer sus contenidos; a esto se le denomina recorrer el árbol binario.

Se distinguen dos formas de recorrido: los que se basan en las relaciones padre-hijo de los nodos, que se denominan recorridos en profundidad, y los que se basan en la distancia o camino desde un nodo cualquiera hasta el nodo raíz, conocidos como recorridos en anchura o por niveles. (Franch, 1999, p. 219)

En este trabajo se mencionan los recorridos por profundidad, por ser los más usados, y son los siguientes:

i. Recorrido en PREORDEN
   - Visitar la raíz
   - Recorrer el subárbol izquierdo
   - Recorrer el subárbol derecho

ii. Recorrido en INORDEN
   - Recorrer el subárbol izquierdo
   - Visitar la raíz
   - Recorrer el subárbol derecho

iii. Recorrido en POSTORDEN
   - Recorrer el subárbol izquierdo
   - Recorrer el subárbol derecho
   - Visitar la raíz.

B. Árboles Binarios de Búsqueda

En un estructura estática de tipo arreglo, la manera más eficiente de realizar una búsqueda es con el algoritmo de búsqueda binaria. Sin embargo, esta estructura presenta la desventaja de no ser eficiente para la inserción y la eliminación de elementos. Por otro lado, una estructura dinámica del tipo lista enlazada tiene mejor comportamiento en las inserciones y eliminaciones de sus elementos, pero no en el algoritmo de búsqueda binaria.

Los arreglos y las listas enlazadas son estructuras lineales debido a que cada elemento de dichas estructuras puede tener a lo mucho un

predecesor y un sucesor. Las desventajas anteriormente mencionadas se deben a esta linealidad. Ante esto, los árboles, representan una opción válida para aprovechar las características positivas de estas estructuras lineales. La propuesta inmediata es el Árbol Binario de Búsqueda.

El árbol binario de búsqueda es una estructura sobre la cual se puede realizar eficientemente la operación de búsqueda, ya que las operaciones de inserción y eliminación se aseguran de que el árbol binario esté optimizado para este fin (Cairó & Guardati, 2002, p. 195).

Para lograr esta eficiencia, es necesario que se cumpla la siguiente condición: "Los Árboles Binarios de Búsqueda son árboles binarios en los cuales se cumple que para cualquier nodo X perteneciente al árbol, todos los datos de los nodos a la izquierda de X son menores que el dato de X y todos los datos de los nodos a la derecha de X son mayores que el dato de X." (Flores, 2005, p. 148)

En la figura 3 se presenta un árbol binario de búsqueda. Se puede verificar que cada uno de los nodos de este árbol cumple con la condición mencionada. Por ejemplo, si se analiza el nodo 120, todos los nodos que se encuentran a su izquierda son menores a 120, y todos los nodos a la derecha de 120 son mayores a él.

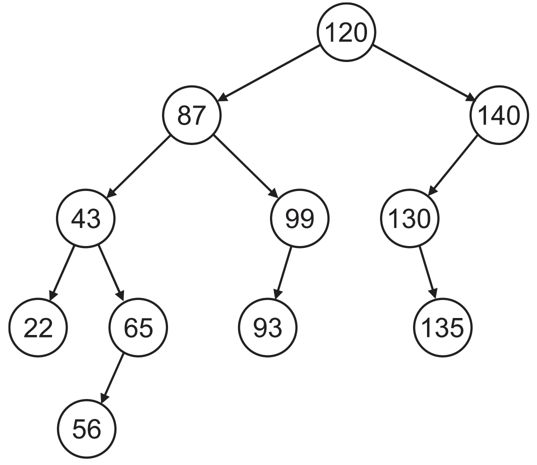


Figura 3. Árbol binario de búsqueda
Adaptado de "Estructura de datos" por Osvaldo Cairo O, 2002, p. 196.

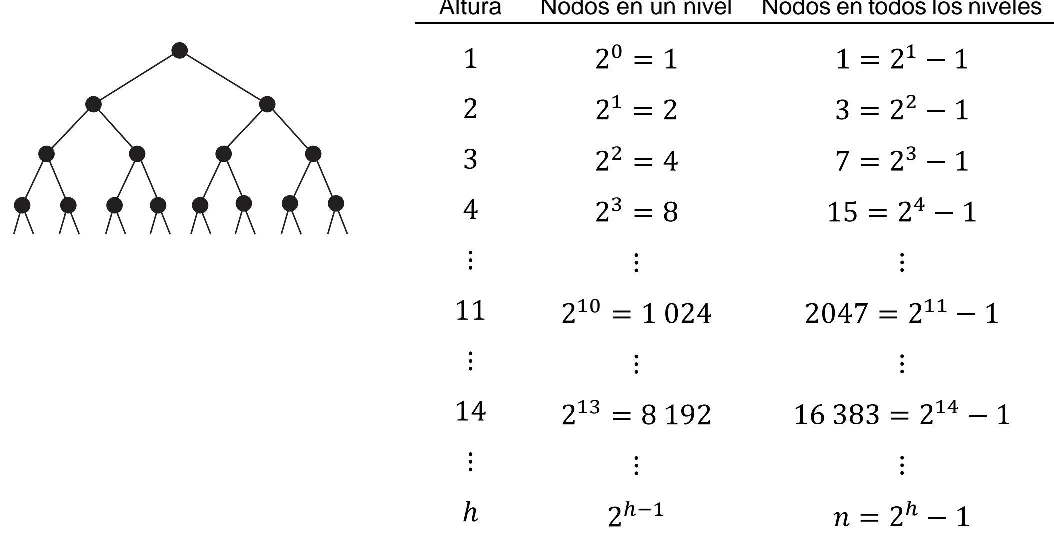
Si se realiza un recorrido inorden del árbol de búsqueda se obtendrá un enlistado de los nodos en forma ascendente. Los diferentes recorridos en el árbol de la figura 3 producen el siguiente resultado:

Preorden: 120 – 87 – 43 – 22 – 65 – 56 – 99 – 93 – 140 -130 – 135
Inorden: 22 – 43 – 56 – 65 – 87 – 93– 99 – 120 – 130 – 135 – 140
Postorden: 22 – 56 – 65 – 43 – 93 – 99 – 87 – 135 – 130 – 140 – 120

C. Árboles Binarios de Búsqueda completos

Los árboles binarios de búsqueda tienen su mayor grado de eficiencia cuando todos sus nodos tienen dos descendientes y cuando los elementos del último nivel están colocados de izquierda a derecha sin dejar

huecos entre ellos. Cuando un árbol binario se organiza de esta forma, se le conoce como árbol completo.



Altura	Nodos en un nivel	Nodos en todos los niveles
1	$2^0 = 1$	$1 = 2^1 - 1$
2	$2^1 = 2$	$3 = 2^2 - 1$
3	$2^2 = 4$	$7 = 2^3 - 1$
4	$2^3 = 8$	$15 = 2^4 - 1$
$\vdots$	$\vdots$	$\vdots$
11	$2^{10} = 1\,024$	$2047 = 2^{11} - 1$
$\vdots$	$\vdots$	$\vdots$
14	$2^{13} = 8\,192$	$16\,383 = 2^{14} - 1$
$\vdots$	$\vdots$	$\vdots$
h	2^{h-1}	$n = 2^h - 1$

Figura 4. Número máximo de nodos en un árbol binario completo.
Adaptado de “Estructura de datos y algoritmos con Java” por Adam Drozdek, 2007, p. 250.

La figura 4 muestra cuántos nodos pueden almacenarse en árboles binarios completos de diferentes alturas. Como cada nodo puede tener dos hijos, el número de nodos en un cierto nivel es el doble del número de padres que residen en el nivel anterior (excepto, por supuesto, la raíz). A partir de esto, se puede deducir que el número de nodos para una altura h es $2^h - 1$.

Si se desea obtener la altura mínima de un árbol binario completo a partir de un total de n nodos, tendremos:

$$\lg(n_h + 1) = h \quad (1)$$

Esto significa que si 10 000 elementos se almacenan en un árbol binario «completamente» balanceado, entonces, se realizarán a lo mucho $\lg(10\,000 + 1) = 13{,}29 \approx 14$ comparaciones para encontrar un elemento en particular. Por otro lado, si 10 000 elementos se almacenan en un arreglo unidimensional o en una lista enlazada, sería necesario realizar a lo mucho 10 000 comparaciones para encontrar un elemento particular. Esto demuestra la superioridad de un árbol frente a una lista.

Por lo tanto, un árbol binario de búsqueda completo minimiza la cantidad de comparaciones para encontrar un dato determinado, lo que se transforma en menos tiempo de búsqueda. Sin embargo, es difícil que un árbol binario de búsqueda pueda mantenerse completo debido a que su procedimiento de inserción no asegura que el árbol crezca siempre de esta forma.

D. Árboles Binarios de Búsqueda auto-balanceados

Hay situaciones en que un árbol binario de búsqueda crece de forma descontrola, por ejemplo, hacia solo uno de sus lados. De esta manera su

eficiencia en la búsqueda disminuye considerablemente. Por ejemplo, en el primer árbol binario de búsqueda de la figura 5, para encontrar un dato, se tendrá que recorrer los nodos de manera secuencial realizando un mayor número de comparaciones

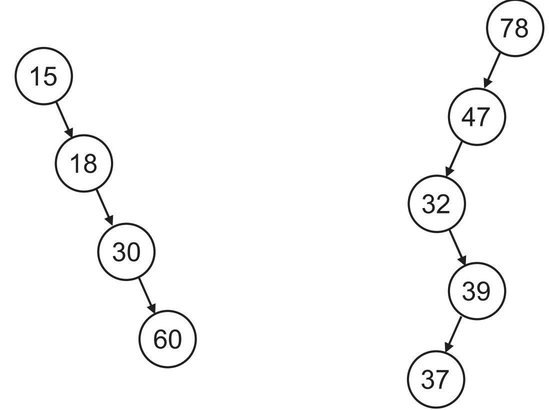


Figura 5. ABB con crecimiento descontrolado.
Adaptado de “Estructura de datos” por Osvaldo Cairo O, 2002, p. 207.

Esta deficiencia se puede evitar si se controla que, mientras se inserten o eliminen datos, el árbol se mantenga casi completo o casi «balanceado» (Ziviani, 2007, p. 156).

Con el objeto de mejorar el rendimiento en la búsqueda surgen este tipo de estructuras. La idea central de estos es la de realizar reacomodos, rotaciones o balances después de las inserciones o eliminaciones de elementos.

Un árbol binario de búsqueda auto-balanceado es un árbol que intenta mantener su altura, o el número de niveles de nodos bajo la raíz, tan pequeños como sea posible en todo momento y automáticamente. Esto es de gran ayuda, ya que las operaciones en un árbol binario de búsqueda tardan en la búsqueda un tiempo proporcional a la altura del árbol, y los árboles binarios de búsqueda ordinarios pueden tomar alturas muy grandes en situaciones normales, como cuando las claves son insertadas en orden creciente o decreciente.

El árbol binario de búsqueda auto-balanceado más común en la literatura es el árbol AVL que se detalla a continuación.

Árboles AVL

Los árboles AVL son Árboles Binarios de Búsqueda auto-balanceados que intentan de mantenerse lo más balanceados posible gracias a sus operaciones de inserción y eliminación. Originalmente se los conocía como árboles admisibles o "*admisible trees*" (Krishnamoorthy, 2008, p. 530) y fueron propuestos en 1962 por los matemáticos G. M. Adelson - Velskii y E. M. Landis en su artículo "*An Algorithm for the organization of information*" (Un algoritmo para la organización de la información).

Formalmente se define un árbol AVL como un árbol binario de búsqueda auto-balanceado en el cual se debe cumplir la siguiente condición: “Para todo nodo del árbol, la altura de la rama derecha e izquierda no debe diferir en más de una unidad”. (Caicedo, Wagner & Méndez, 2010, p. 111)

La condición anterior es la que define al término «factor de equilibrio».

$$FE = H_{RD} - H_{RI} \tag{2}$$

Por lo tanto, el factor de equilibro de todos los nodos de un Árbol AVL puede ser -1, 0 o +1. (Gomez & Ania, 2008, p. 104)

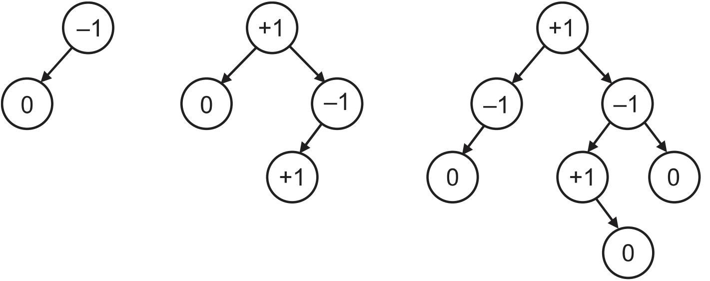


Figura 6. Ejemplos de Árboles AVL

Adaptado de “Estructura de datos” por Osvaldo Cairo O, 2002, p. 208.

En la figura 6 se muestra una representación gráfica de árboles AVL. Dentro de cada nodo se observa su factor de equilibrio (FE).

Los árboles AVL facilitan el manejo del balanceo (Langsam, 1996), pues solo se tiene en cuenta la diferencia de alturas de los sub-árboles derecho e izquierdo; es decir que el balanceo del árbol puede realizarse localmente si solo afecta a una porción del árbol cuando se requieren cambios después que se inserte o elimine un elemento en el árbol.

Análisis de complejidad de un árbol AVL

En esta sección se muestra un análisis de cuánto puede crecer como máximo un árbol AVL, es decir, cuál será su altura en el peor de los casos. Este dato servirá para tener una idea del costo de las operaciones de búsqueda, inserción y eliminación.

Según la expresión (2), se define el factor de equilibrio como la altura de la rama derecha menos la altura de la rama izquierda. Entonces, en un árbol AVL, todos los nodos cumplen la propiedad de que el valor del factor de equilibrio sea: -1, 0 o +1.

Sea n_h el mínimo número de nodos en un árbol AVL de una altura h dada, que se encuentra en su peor caso de desbalance; si se agregase un nodo, tal que la nueva altura sea $(h + 1)$, deja de cumplir su condición de equilibrio y por lo tanto, deja de ser AVL.

Tabla 1

Árboles Fibonacci, hasta h=4

$\boldsymbol{h}$	$\boldsymbol{n_h}$	FE DE LOS NODOS
0	$n_1 = 0$	
1	$n_1 = 1$	0
2	$n_2 = 2$	1, 0
3	n_3 $= n_2 + n_1 + 1$ $n_3 = 4$	1, 0, 1, 0 — 1, 0, -1, 0
4	n_4 $= n_3 + n_2 + 1$ $n_4 = 7$	1, 1, 0, 1, 0, 1, 0

Fuente: Elaboración propia

Interpretación:

La tabla 1 muestra árboles AVL en su peor caso de desbalance, también denominados “árboles de Fibonacci”, y los factores de equilibrio de sus nodos, para alturas 0; 1; 2; 3 y 4. Se muestran los casos desbalanceados por la derecha, porque los de la izquierda son especulares.

Se cumple que: $n_3 = n_2 + n_1 + 1$, entonces $n_3 = 4$.

Se observa que se pueden generar 2 árboles por la derecha y 2 árboles por la izquierda (especulares) con altura tres ($h = 3$).

Se observa que $n_4 = 7$ porque: $n_4 = n_3 + n_2 + 1$.

En la figura 7, a la raíz se agrega por la derecha un árbol de Fibonacci de altura 4, y por la izquierda uno de altura 3.

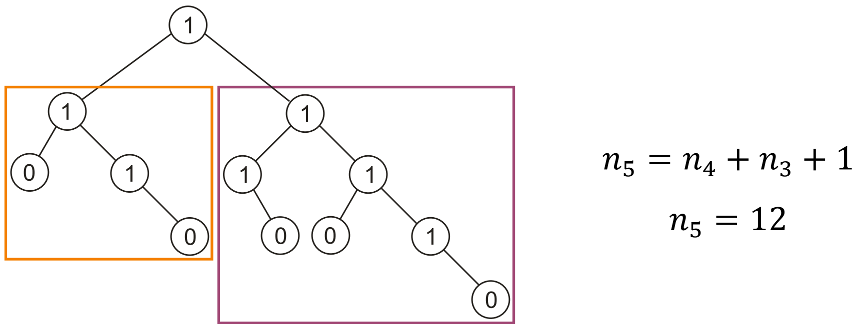


Figura 7. Árbol Fibonacci para h=5

Fuente: Elaboración propia

Luego se puede asegurar que el total de nodos para $h = 5$ es $n_5 = 12$ porque $n_5 = n_4 + n_3 + 1 = 7 + 4 + 1$.

Por lo anterior, mediante inducción se puede decir que, en general, se tiene la recurrencia:

$$n_h = n_{h-1} + n_{h-2} + 1 \quad (3)$$

Donde $n_0 = 0$ y $n_1 = 1$ son las condiciones iniciales.

Al añadir 1 a ambos lados de la expresión (3), se obtiene:

$$n_h + 1 = (n_{h-1} + 1) + (n_{h-2} + 1)$$

Así, los números $n_h + 1$ son números de Fibonacci ($F_h = F_{h-1} + F_{h+2}$) (Puntambekar, 2009, pp. 11-4)

Tabla 2

Secuencia números de Fibonacci

h	0	1	2	3	4	5	6	7
n_h	0	1	2	4	7	12	20	33
F_h	0	1	1	2	3	5	8	13
F_{h+2}	1	2	3	5	8	13	21	34

Fuente: Elaboración propia

En la tabla 2 se observan las dos secuencias de números que generan n_h y F_h. De estas secuencias se obtiene que:

$$n_h = F_{h+2} - 1 \qquad (4)$$

Usando la fórmula Moivre[1] para los números Fibonacci se tendrá:

$$F_h = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^h - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^h$$

[1]Fórmula para obtener el término n-ésimo de la sucesión de Fibonacci, es también llamada por otros autores como fórmula de Binet.

$$F_{h+2}=\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{h+2}-\frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^{h+2}$$

Luego, reemplazando la expresión anterior en la expresión (4), se tiene:

$$n_h=\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{h+2}-\frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^{h+2}-1$$

Debido a que $\left|\frac{1}{2}\left(1-\sqrt{5}\right)\right|\approx-0{,}618034$, el segundo término de esta ecuación rápidamente disminuye con el aumento de h y tiene el valor máximo de 0,17082 para $\mathrm{h}=0$, por consiguiente,

$$n_h\geq\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{h+2}-0{,}17082-1\geq\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{h+2}-2$$

$$n_h+2\geq\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{h+2}\qquad(5)$$

Al aplicar logaritmo binario a ambos lados de la expresión (5), se obtiene:

$$\lg(n_h+2)\geq\lg\frac{1}{\sqrt{5}}+2\lg\left(\frac{1+\sqrt{5}}{2}\right)+h\times\lg\left(\frac{1+\sqrt{5}}{2}\right)$$

$$\lg(n_h+2)\geq0{,}22787+0{,}69424h\qquad(6)$$

A partir de (6) se obtiene el límite superior en h

$$h \leq 1{,}44042 \times \lg(n_h + 2) - 0{,}32824 \tag{7}$$

Luego se acota h con las expresiones (1) y (7):

$$\lg(n_h + 1) \leq h \leq 1{,}44042 \times \lg(n_h + 2) - 0{,}32824 \tag{8}$$

Entonces la altura del peor caso de un árbol AVL con n nodos es aproximadamente $1{,}44 \times \lg(n_h + 2) - 0{,}32$.

Si 10 000 elementos se almacenan en un árbol AVL, entonces el árbol, en el peor de los casos, tiene una altura de $1{,}44 \times \lg(10\,002) - 0{,}32 = 18{,}81 \approx 19$. Esto significa que si 10 000 elementos se almacenan en un árbol AVL, entonces se revisarán, en el peor de los casos, 19 nodos para encontrar un elemento en particular. Un árbol completo con la misma cantidad de nodos, usando la expresión (1), tendría una altura de 14, por consiguiente, el tiempo de búsqueda en el peor caso en un árbol AVL es 44 % peor (requiere 44 % más comparaciones) que en la configuración de árbol en el mejor de los casos (Knuth, 2000).

Las operaciones de inserción y eliminación en un árbol AVL son parecidas a las de un árbol binario de búsqueda convencional, con la diferencia que, en cada operación, la estructura debe asegurarse que se

mantenga el factor de equilibrio de todos los nodos en 0 o ±1, para esto realizará reacomodos o rebalanceos cuando sea necesario.

Respecto a la operación de inserción de nodos en un árbol AVL, las pruebas experimentales indican que el rebalanceo es necesario cada dos inserciones, siendo igualmente probables las rotaciones simples y dobles (Drozdek, 2007). Es decir que un árbol AVL es tan eficiente, en la inserción, como un árbol binario de búsqueda convencional.

En relación con la operación de eliminación, se debe resaltar el hecho de que mientras la inserción de un nodo puede producir como máximo una rotación, cuando se elimina un nodo, la eliminación puede necesitar una rotación en cada nodo de la trayectoria de búsqueda (Cairó & Guardati, 2002). Este dato puede resultar alarmante. Sin embargo, los resultados empíricos muestran que sólo es necesario el rebalanceo en una de cada cinco eliminaciones, es decir, el 75 % de los casos de eliminación no requieren balanceo en lo absoluto. Por otra parte, sólo el 53 % de las inserciones no sacan al árbol de balance (Karlton, 1976). Es decir, la eliminación que consume más tiempo, ocurre con menos frecuencia.

Árboles Red–Black

Un árbol coloreado, rojinegro o *Red–Black* es un árbol binario de búsqueda auto-balanceado. La estructura original fue creada por Rudolf

Bayer en 1972, que le dio el nombre de “árboles-B binarios simétricos”, pero tomó su nombre moderno en un trabajo de Leo J. Guibas y Robert Sedgewick realizado en 1978.

La característica más resaltante de un árbol *Red–Black*, y que lo diferencia de otros tipos de árbol, es que cada uno de sus nodos tiene asociado un color, el cual puede ser rojo o negro.

En la memoria, cada nodo almacenará un campo adicional de información que indicará su color. Sobre este campo «color» se aplican restricciones que resultan en un árbol en el que ningún camino de la raíz a una hoja es más de dos veces más largo que cualquier otro, lo cual asegura que el árbol se mantenga balanceado (Silva, 2017). Cada nodo de un árbol coloreado contiene la siguiente información: color, clave, hijo izquierdo, hijo derecho y padre. Si un hijo de un nodo no existe, el apuntador correspondiente contiene el valor `NULL`, el cual se considerará como un nodo externo cuyo color es negro.

Todo árbol *Red–Black* satisface las siguientes propiedades:

1. Todo nodo es rojo o negro
2. La raíz es negra
3. Todo nodo externo (`NULL`) es negro.

4. Si un nodo es rojo, entonces sus dos hijos son negros.
5. Para cada nodo, todas las rutas de un nodo a los nodos externos (`NULL`) contienen el mismo número de nodos negros.

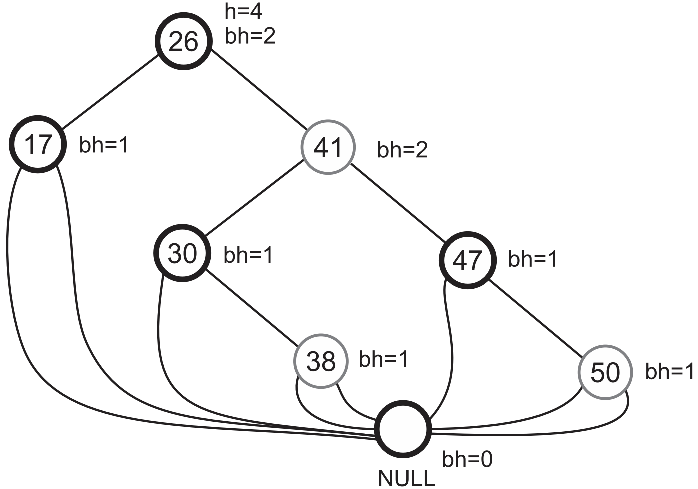


Figura 8. Árbol Red–Black.
Adaptado de “Árboles Rojo-Negros” por Guido Urdaneta, 2010, p. 3.

En la figura 8 se tiene un árbol *Red–Black* donde los nodos de color negro están representados con un borde más ancho que el de los nodos de color rojo. Debe notarse que el árbol cumple la propiedad de un árbol binario de búsqueda, además de cumplir todas las propiedades de un árbol *Red–Black* mencionadas anteriormente.

Dado que las operaciones básicas como insertar, borrar y buscar un elemento tienen un peor tiempo de respuesta proporcional a la altura del árbol, la cota superior de la altura permite a los árboles *Red–Black* ser

eficientes en el peor caso, de forma contraria a lo que sucede en los árboles binarios de búsqueda. Para ver que estas propiedades garantizan lo dicho, basta ver que ningún camino puede tener 2 nodos rojos seguidos debido a la propiedad 4. "El camino más corto posible tiene todos sus nodos negros, y el más largo alterna entre nodos rojos y negros" (Cormen, Leiserson, Rivest, & Stein, 2009, p. 309). Como todos los caminos máximos tienen el mismo número de nodos negros (por la propiedad 5), esto muestra que no hay ningún camino que pueda tener el doble de longitud que otro camino.

Los árboles *Red–Black* ofrecen un peor caso con tiempo garantizado para la inserción, el borrado y la búsqueda. No es esto únicamente lo que los hace valiosos en aplicaciones sensibles al tiempo como las aplicaciones en tiempo real, sino que además son apreciados para la construcción de bloques en otras estructuras de datos que garantizan un peor caso (Urdaneta, 2010). Por ejemplo, muchas estructuras de datos usadas en geometría computacional pueden basarse en árboles *Red–Black*.

El árbol AVL es un tipo de estructura con $\Theta(\lg n)$ tiempo de búsqueda, inserción y borrado. Está equilibrado de forma más rígida que los árboles *Red–Black*, lo que provoca que la inserción y el borrado sean

más lentos, pero la búsqueda y la devolución del resultado de la misma sean más veloz.

En un árbol *Red–Black*, con n nodos internos se cumple la siguiente condición:

$$h \leq 2 \times \lg(n+1) \tag{9}$$

Se define la función altura negra de un nodo x, $bh(x)$, como la cantidad de nodos negros de cualquier camino desde x hasta un nodo externo, no contando el nodo x. Se probará mediante inducción, que un subárbol, que parte en el nodo x, contiene $2^{bh(x)} - 1$ nodos internos como mínimo.

Si x es un nodo externo, entonces $bh(x) = 0$, y el número de nodos internos es: $2^0 - 1 = 0$.

Si x es un nodo hoja, entonces $bh(x) = 1$, y el número de nodos internos es: $2^1 - 1 = 1$.

Si x tiene una altura h, los nodos hijos de x, tienen altura (h-1).

- Si el nodo hijo es rojo, entonces, tiene altura negra: $bh(x)$, igual a la de su padre.

- Si el nodo hijo es negro, entonces, tiene altura negra: $bh(x) - 1$, ya que no se cuenta el nodo negro.

Considerando verdadera la proposición para el número de nodos internos del subárbol que comienza en x; los subárboles de los hijos de x, deben entonces, tener a lo menos: $2^{bh(x)} - 1$ nodos internos.

Para obtener el número de nodos internos del subárbol que comienza en x, se suman los nodos internos de los subárboles hijos, más el nodo interno x; obteniéndose:

$$
\begin{gathered}
n \geq \left(2^{bh(x)-1} - 1\right) + \left(2^{bh(x)-1} - 1\right) + 1 \\
n \geq 2 \times \left(2^{bh(x)-1}\right) - 1 \\
n \geq 2^{bh(x)} - 1
\end{gathered}
\tag{10}
$$

Pero en un árbol *Red–Black* al menos la mitad de los nodos de un camino de un nodo hasta los nodos hoja deben ser negros, entonces, si h es la altura de un árbol que comienza en x, se tiene que:

$$bh(x) \geq \frac{h}{2}$$

Reemplazando en la expresión (10), en función de $bh(x)$, la cota para $bh(x)$, se obtiene:

$$n \geq 2^{bh(x)} - 1$$

$$n \geq 2^{h/2} - 1$$

Despejando h, se logra:

$$h \leq 2 \times \lg(n + 1) \qquad (11)$$

De esta forma se comprobó la expresión (9). Debe tenerse en cuenta que ésta es la complejidad de un árbol *Red–Black* en el peor caso.

Al insertar o descartar nodos en un árbol *Red–Black*, pueden violarse las propiedades que los definen; y para mantenerlo coloreado, deben cambiarse los colores de algunos nodos y también posiblemente efectuar rotaciones. Esto se refleja en un costo adicional para las funciones de inserción y descarte en un árbol binario de búsqueda.

Para un árbol AVL, la cota para la altura resulta menor que en un árbol *Red–Black*:

$$h_{AVL} \leq 1{,}44 \times \lg(n + 2) - 0{,}32 = \Theta(\lg n) \qquad (12)$$

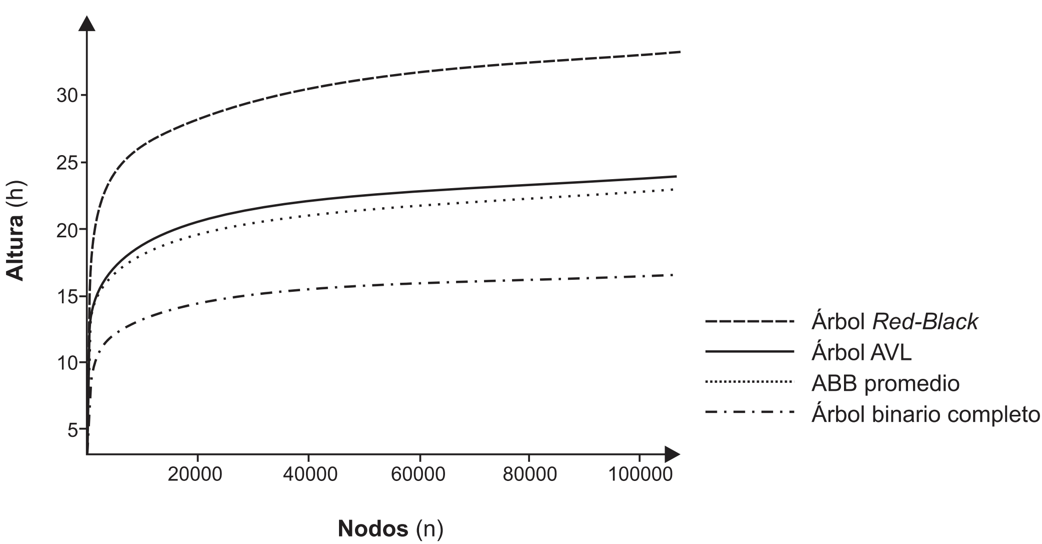


Figura 9. Complejidad de las alturas de árboles.
Adaptado de "*Red Black Trees*" por Leopoldo Silva, 2008, p. 4.

La figura 9 muestra la complejidad del árbol *Red–Black*, respecto del AVL, y puede observarse que son muy similares. Pero las operaciones se realizan en menor tiempo en un árbol *Red–Black*. En un árbol binario de búsqueda, en promedio, si las claves llegan aleatoriamente, se tiene la altura: $h_{ABB} \leq 1{,}3863 \times \lg n = \Theta(\lg n)$. Pero esta se incrementa hasta n, en el peor caso.

Árboles Splay

Un árbol biselado, desplegado o *splay* es un árbol binario de búsqueda auto-balanceado que emplea rotaciones para mover una clave en particular, ya sea en búsqueda, inserción o eliminación, hacia la raíz.

Esto hace que los nodos que se han accedido recientemente se ubiquen cerca de la raíz, haciendo que la búsqueda de ellos sea más eficiente.

No es necesario almacenar información adicional en el nodo, ya sea el factor de balance (como en los árboles AVL), o el color (como en los árboles *Red–Black*). La forma del árbol va variando de acuerdo a los nodos que son más recientemente accedidos.

Este tipo de árbol fue desarrollado por Sleator y Tarjan en 1985, en la publicación del ACM Journal, "*Self-organizing Binary Search Trees*" como una alternativa a los algoritmos que mantienen balanceado un árbol binario de búsqueda.

Una opción conservadora para organizar los nodos, es adelantar en una posición el elemento buscado, cada vez que hay un acceso a esa clave; otra, más enérgica, es llevar el elemento al inicio de la lista. Puede comprobarse que mover al frente tiene un mejor comportamiento, en caso de distribuciones de búsqueda que cambian.

"Si algo ha sido accesado, es muy probable que sea nuevamente accesado." (Cortez & Vega, 2009, p. 41). En el caso de los árboles biselados, se lleva el elemento buscado o insertado a la posición de la raíz.

Uno de los peores casos para el algoritmo básico del árbol biselado es el acceso secuencial a todos los elementos del árbol de forma ordenada. Esto deja el árbol completamente desbalanceado (son necesarios n accesos, cada uno de los cuales del orden de $\Theta(\lg n)$ operaciones). Volviendo a acceder al primer elemento se dispara una operación que toma el orden de $\Theta(n)$ operaciones para volver a balancear el árbol antes de devolver este primer elemento.

Esto es un retraso significativo para esa operación final, aunque el rendimiento se amortiza si se tiene en cuenta la secuencia completa es del orden de $\Theta(\lg n)$. Sin embargo, investigaciones recientes muestran que si aleatoriamente se vuelve a balancear el árbol se podrá evitar este efecto de desbalance y dar un rendimiento similar a otros algoritmos de auto-balanceo.

Como se puede observar en la figura 10, al insertar unos datos en forma ascendente se va construyendo un árbol que tiene forma lineal, lo cual es una forma ineficiente para realizar búsquedas. Lo particular de este tipo de árbol es que conforme se vayan realizando búsquedas e inserciones, el árbol se auto-balancea ubicando el nodo buscado o insertado en la raíz del árbol.

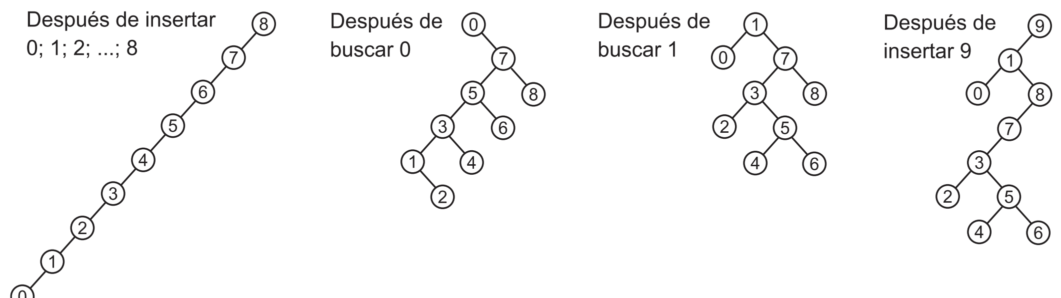


Figura 10. Operaciones en un árbol splay.
Adaptado de "*Splay Trees*" por Leopoldo Silva, 2008

Los árboles biselados se comportan de forma parecida que los AVL, con la diferencia en que en los AVL interesa más mantener el equilibrio (la diferencia de cero o uno entre las alturas de las ramas de un nodo en particular), mientras que en los árboles biselados no es tan importante el costo de un acceso individual, que puede ser tan malo como $\Theta(n)$, en su lugar se desea asegurar que este tipo de comportamiento no pueda producirse repetidamente. (Cortez & Vega, 2009, p. 45)

Este tipo de estructura es eficiente para implementar caches o algoritmos de recolección de basura, en donde se buscan generalmente los nodos más frecuentes. En el caso que se haga un acceso uniforme a los datos, el rendimiento de un árbol *splay* será considerablemente peor que un árbol binario de búsqueda simple.

2.2.2. Búsqueda de datos

La búsqueda (*searching*) de datos está relacionada con las tablas para consultas (*lookup*). Estas tablas contienen una cantidad de información que se almacena en forma de listas de parejas de datos (Joyanes, 2008). Por ejemplo, un diccionario con una lista de palabras y definiciones; un catálogo con una lista de libros de informática; una lista de estudiantes y sus notas; un índice con títulos y contenido de los artículos publicados en una determinada revista, etc. En todos estos casos es necesario con frecuencia buscar un elemento en un conjunto de datos.

Según Joyanes (2008) la búsqueda por claves para localizar registros es, con frecuencia, una de las acciones que mayor consumo de tiempo conlleva y, por consiguiente, el modo en que los registros están dispuestos y la elección del modo utilizado para la búsqueda pueden redundar en una diferencia sustancial en el rendimiento del programa.

Un algoritmo de búsqueda es un algoritmo que acepta un argumento x y trata de encontrar un registro cuyo valor es x.

Como la forma en la que están organizados los datos pueden ser arreglos, una lista enlazada, un árbol, o inclusive un grafo; es que existen diferentes técnicas de búsqueda que pueden ser apropiadas para diferentes organizaciones. Una estructura generalmente está diseñada

para alguna técnica de búsqueda específica. La estructura puede estar contenida en forma completa en memoria interna, completamente en una memoria auxiliar, o puede estar partida entre los dos. En este caso, se requieren diferentes técnicas de búsqueda de acuerdo a las diferentes condiciones que se asuman. Aquellos tipos de búsqueda en las cuales la estructura está completamente contenida en la memoria interna, son llamados búsquedas internas, mientras que aquellas en las cuales la estructura está almacenada en una memoria auxiliar son denominadas búsquedas externas (Tenenbaum, 2004, p. 437).

Búsqueda secuencial

La búsqueda secuencial se da generalmente en estructuras estáticas del tipo arreglo o en estructuras dinámicas del tipo lista enlazadas. Es el método más óptimo para buscar un elemento en una estructura lineal. Se explora secuencialmente el arreglo o lista o, dicho en otras palabras, se recorre la estructura lineal desde el primer elemento al último. Si se encuentra el elemento buscado, se debe visualizar un mensaje similar a «Fin de búsqueda»; en caso contrario, visualizar un mensaje similar a «Elemento no encontrado».

En otras palabras, la búsqueda secuencial compara cada elemento de la estructura con el valor deseado, hasta que éste encuentra o termina de leer el total de los elementos de la estructura completa.

Como la búsqueda es una actividad tan común en computación, es conveniente encontrar un método eficiente para ejecutarla. El método de búsqueda menos refinado es el de búsqueda lineal o secuencial. Si la lista está desordenada y construida al azar, la búsqueda lineal puede ser la única vía de encontrar algo en ella (a no ser, que se ordene primero). Sin embargo, no debería usarse este para buscar un nombre en un directorio telefónico. En su lugar, se abre el libro en una página cualquiera y se examinan los nombres que aparecen en ella. Como están ordenados en forma alfabética, dicho examen determinará si debe continuarse la búsqueda en la primera o segunda mitad del libro.

El método de búsqueda más eficiente para estructuras lineales es la búsqueda binaria. Básicamente se compara el argumento con la llave del elemento medio de la lista. Si son iguales, la búsqueda termina con éxito; en caso contrario, se busca de manera similar en la mitad izquierda o derecha de la lista (Tenenbaum, 2004, p. 398), pero para aplicar este método, es necesario que la estructura lineal tenga sus elementos previamente ordenados.

Búsqueda en árboles binarios de búsqueda

Se debe recordar que los árboles binarios de búsqueda tienen una definición particular: para cada nodo X, las claves de su subárbol izquierdo deben ser menores que la clave del nodo X y las claves de su subárbol derecho deben ser mayores que la clave del nodo X.

Esta propiedad de los árboles binarios de búsqueda, aunque eleve el costo de insertar nuevos elementos, facilita grandemente la búsqueda de datos (Cairó & Guardati, 2002). Si el valor buscado no es igual al nodo actual, sólo existen dos posibilidades: que sea mayor o que sea menor. Lo que implica que el nodo buscado puede pertenecer a uno de los dos subárboles. Cada vez que se toma la decisión de buscar en uno de los subárboles de un nodo, se están descartando los nodos del otro subárbol. En el caso de los árboles binarios de búsqueda completos, se descarta la mitad de los elementos de la estructura, esto cumple el modelo: $T(n) = T(n/2) + c$, lo cual asegura un costo logarítmico.

Si $T(h)$ es la complejidad de la búsqueda en un árbol de altura h, en cada iteración, el problema se reduce a uno similar pero con la altura disminuida en uno.

Entonces:

$T(h) = T(h-1) + \Theta(1)$ con $T(0) = 0$

La solución de esta recurrencia, es:

$T(h) = h \times \Theta(1) = \Theta(h)$

Pero en árboles binarios de búsqueda se tiene que: $\log_2 n \leq h \leq n$ Entonces: $\Theta(\log_2 n) \leq T(h) \leq \Theta(n)$.

2.3. Definición de términos

Árbol auto-balanceado: Un árbol binario de búsqueda auto-balanceado es aquel que además de ser homogéneo, se caracteriza porque intenta mantener su altura, o el número de niveles de nodos bajo la raíz, tan pequeños como sea posible en todo momento.

Árbol AVL: Árbol binario de búsqueda auto-balanceado propuesto por Adelson - Velskii y Landis con una condición de balanceo estricta conocida como “factor de equilibrio”.

Búsqueda de datos: La búsqueda de datos en un ordenador se realiza a través de algoritmos que estén diseñados para localizar un elemento con ciertas propiedades dentro de una estructura de datos; por ejemplo, ubicar el registro correspondiente a cierta persona en una base de datos, o el mejor movimiento en una partida de ajedrez.

DASA: Dirección Académica de Actividades y Servicios Académicos de la Universidad Nacional Jorge Basadre Grohmann.

Número de comparaciones: La eficiencia de los algoritmos se mide por el número de comparaciones e intercambios que tienen que hacer, es decir, se toma n como el número de elementos que tiene el arreglo o vector

a ordenar y se dice que un algoritmo realiza $\Theta(n^2)$ comparaciones cuando compara n veces los n elementos, $\mathrm{n} \times \mathrm{n} = n^2$

Tiempo de respuesta: También conocido como tiempo de ejecución, se refiere al intervalo de tiempo en el que un programa de computadora se ejecuta en un sistema operativo.

UNJBG: Universidad Nacional Jorge Basadre Grohmann.

Velocidad de respuesta: Es la velocidad con la que se otorga una respuesta completa ante la solicitud de un procedimiento específico.

CAPÍTULO III

MARCO METODOLÓGICO

3.1. Tipo y diseño de la investigación

El diseño de la presente investigación es no experimental – transversal de nivel descriptivo.

3.2. Población y muestra

La población de datos, contenida en los registros de una base de datos, está conformada por un total de 6 966 códigos de estudiantes bajo un formato especial definido por el sistema de registro académico de la Universidad Nacional Jorge Basadre Grohmann de Tacna. Se tomó esta cantidad de registros en base al número total de estudiantes matriculados en el periodo académico 2017 - I. De esta población se tomarán muestras que se calcularán de la siguiente manera ya que se trata de una población finita:

$$n = \frac{N \times Z^2 \times p \times q}{e^2 \times (N-1) + Z^2 \times p \times q} \qquad (13)$$

Los datos a considerar son los siguientes:

N: Tamaño de la población 6 966

p: 50 % probabilidad a favor.

q: (1- p) = 50 % probabilidad en contra.

Z: Para un nivel de confianza del 95 % su valor es 2

e: 10 % = 0,1

$$n = \frac{6\,966 \times 2^2 \times 0{,}5 \times 0{,}5}{0{,}1^2 \times (6\,966 - 1) + 2^2 \times 0{,}5 \times 0{,}5} = \frac{6\,966}{70{,}65} = 98{,}60$$

Entonces, el tamaño de la muestra que se debe considerar es de 99 registros.

3.3. Operacionalización de variables

Variable dependiente: Velocidad de respuesta en la búsqueda de datos.

Variable independiente: Estructuras de datos dinámicas.

Variable	Definición conceptual	Definición operacional	Indicadores
Velocidad de respuesta en la búsqueda de datos	Es la velocidad con la que se otorga una respuesta completa ante la solicitud de un	Tiempo requerido para encontrar un dato en memoria.	• Tiempo de respuesta
		Número de comparaciones efectuadas en	• Número de comparaciones

Variable	Definición conceptual	Definición operacional	Indicadores
	procedimiento de búsqueda de datos en memoria.	una estructura para encontrar un dato en memoria.	
Estructuras de datos dinámicas	Estructuras de datos en las que su tamaño en memoria puede crecer o contraerse a medida que se ejecuta el programa	Colección de claves en nodos que definen la altura de la estructura, manteniendo el equilibrio mediante rebalanceo.	• Altura • Número de rotaciones

3.4. Técnicas e instrumentos para recolección de datos

Para la presente investigación se recogieron datos provenientes de los resultados de las operaciones de búsqueda realizadas a partir de los 6 966 registros de los estudiantes matriculados en el periodo 2017 – I contenidos en un archivo de texto CSV y que fueron migrados a dos estructuras de datos dinámicas del tipo árbol binario de búsqueda auto-balanceado almacenadas en memoria principal del computador.

3.5. Procesamiento y análisis de datos

Para tratar los datos se hizo uso de la estadística descriptiva y las pruebas de contrastación de hipótesis. Para ello se utilizó el software de tratamiento estadístico “Minitab” en su versión 16.

Los datos almacenados en las estructuras de datos dinámicas contienen un campo de datos denominado `código` y otro campo denominado `especialidad`.

El campo `código` está formado por diez caracteres: los cinco primeros caracteres están referidos al año y semestre de ingreso, el sexto y sétimo carácter es el identificador de la “especialidad” a la que se matriculó el estudiante y los otros tres caracteres se forman por el correlativo de las fichas de matrícula del registro de postulantes en cada especialidad.

El campo `especialidad` es un código relacionado a cada una de las especialidades a las que los alumnos se pueden matricular; estos códigos son los que se usan en la base de datos del sistema de registro académico de la Dirección Académica de Actividades y Servicios Académicos (DASA). La relación de estos códigos se muestran en la tabla 4.

Tabla 3

Códigos de las especialidades de la UNJBG

Código	Especialidad
1	Ingeniería de Minas
2	Ingeniería Geológica - Geotecnia
3	Ingeniería Civil
4	Ciencias Contables y Financieras
5	Ingeniería Metalúrgica
6	Ingeniería Comercial
7	Ingeniería Mecánica
8	Arquitectura
9	Ciencias Administrativas
10	Ingeniería Pesquera
11	Ciencias de la Comunicación
12	Agronomía
13	Enfermería
14	Economía Agraria
15	Medicina Veterinaria y Zootecnia
16	Educación - Ciencias de la Naturaleza y Promoción Educativa Ambiental
17	Odontología
18	Ingeniería en Industrias Alimentarias
19	Farmacia y Bioquímica
20	Educación - Idioma Extranjero
21	Educación - Lengua, Literatura
22	Obstetricia
23	Medicina Humana
24	Educación - Ciencias Sociales y Promoción Socio Cultural
25	Educación - Lengua, Literatura y Comunicación Intercultural
26	Educación - Matemática, Computación e Informática

Código	Especialidad
27	Educación - Idioma Extranjero, Traductor e Interprete
28	Biología - Microbiología
29	Ingeniería en Informática y Sistemas
30	Ingeniería Química
31	Física Aplicada
32	Física Aplicada - Especialidad en Energías Renovables
33	Derecho y Ciencias Políticas
34	Artes
35	Programa Académico de Administración de Empresas
36	Programa Académico de Agronomía
37	Educación - Especialidad en Idioma Extranjero
38	Educación - Lengua, Literatura y Gestión Educativa
39	Educación - Ciencias de la Naturaleza, Tecnología y Ambiente
40	Física Aplicada - Especialidad en Electrónica
41	Matemática
42	Ingeniería Ambiental
43	Historia

Fuente: Base de datos del Sistema Académico – DASA.

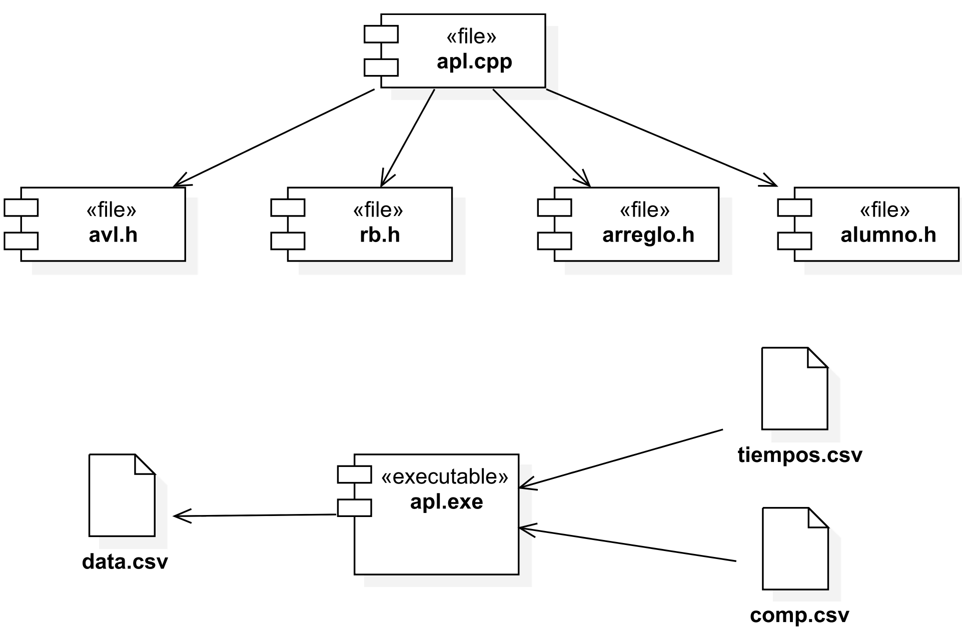


Figura 11. Diagrama de componentes de la aplicación desarrollada

Fuente: Elaboración propia

Se desarrolló una aplicación informática utilizando el lenguaje de programación C++ bajo el paradigma orientado a objetos. En la figura 11 se visualiza el diagrama de componentes de la aplicación desarrollada. En la parte superior están los ficheros fuente desarrollados, y en la parte inferior está el ejecutable `apl.exe` que hace uso de la fuente de datos `data.csv`, que contiene los registros de los estudiantes matriculados en el periodo 2017 – I, y los ficheros de salida `tiempos.csv` y `comp.csv` donde se almacenan los tiempos de búsqueda y número de comparaciones en los ficheros respectivamente.

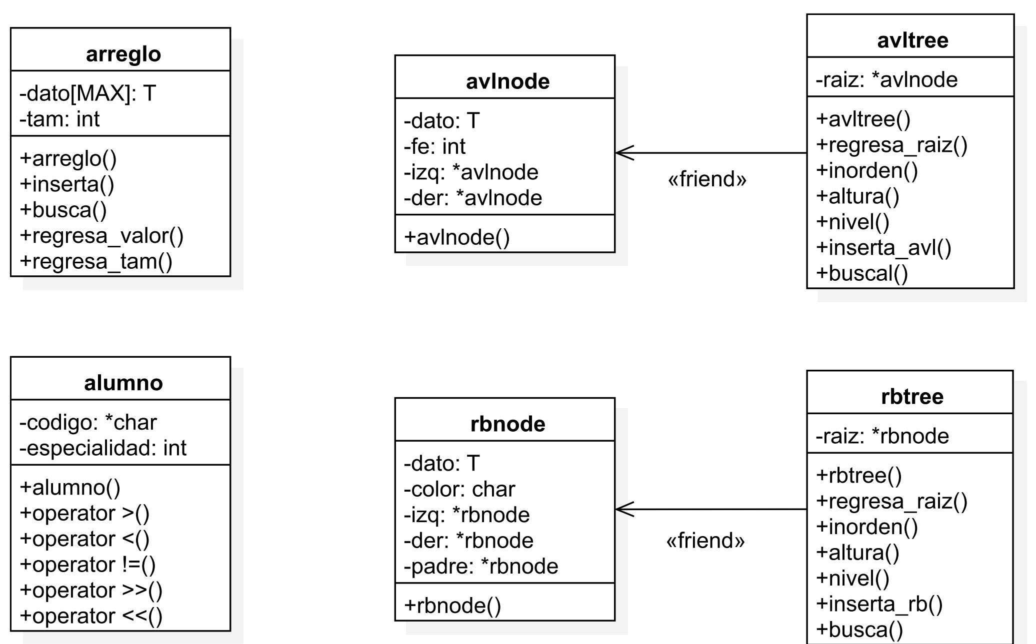


Figura 12. Diagrama de clases de la aplicación desarrollada

Fuente: Elaboración propia

En la figura 12 se visualizan las clases abstraídas para el desarrollo de la aplicación, se debe resaltar las clases `avlnode` y `rbnode` son las que representan las estructuras de los nodos de los árboles AVL y *Red–Black* respectivamente; y que estos se diferencian por el dato `fe` que representa el factor de equilibro de cada nodo (en el caso de los árboles AVL) y por el dato `color` que representa el color (rojo o negro) de cada nodo (en el caso de los árboles *Red–Black).*

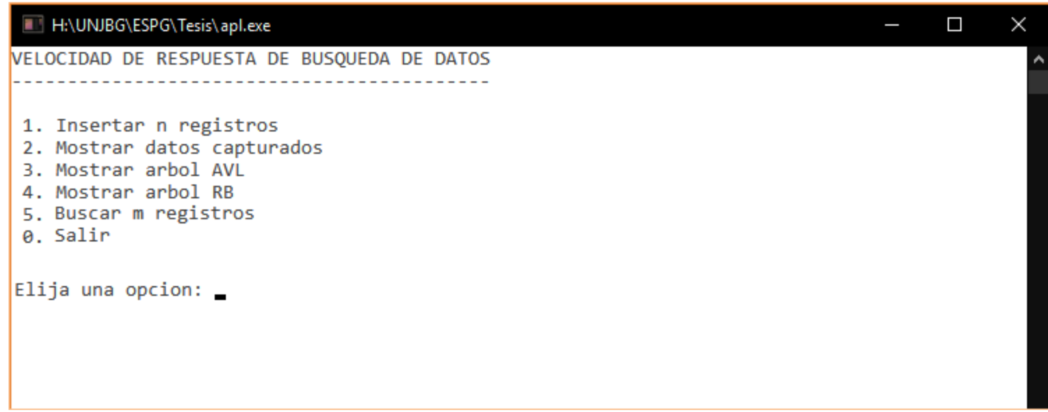


Figura 13. Menú del programa utilizado para generar los registros

Fuente: Elaboración propia

La interfaz de la aplicación implementada se muestra en la figura 13. Con la opción 1 se extraen los 6 966 registros que están almacenados en el archivo `data.csv`, generado a partir de la base de datos del Sistema Académico de la DASA, y son almacenandos en la memoria principal (específicamente en las estructuras de datos dinámicas) los códigos y la especialidad a la que se matriculó el estudiante.

CAPÍTULO IV

RESULTADOS

En base a los 6 966 registros de los estudiantes matriculados en el periodo 2017 – I de la UNJBG, se extraen los códigos del estudiante que consta de 10 caracteres y el identificador de la especialidad a la que se matriculó. En la figura 14 se muestra los registros de la población que están almacenados en un archivo de texto CSV para el posterior procesamiento y análisis estadístico.

	A	B	C
1	N°	Codigo	Especialidad
2	1	19772145	9
3	2	19919611	4
4	3	199310839	4
5	4	199310940	9
6	5	199412265	12
7	6	199412782	29
8	7	199614373	4
9	8	199715614	14
10	9	199715859	29
11	10	199816106	3
12	11	199917454	9
⋮	⋮	⋮	⋮
6963	6962	2017130047	2
6964	6963	2017130048	2
6965	6964	2017130049	2
6966	6965	2017130050	2
6967	6966	2017130051	2

Figura 14. Archivo de texto CSV con 6 966 registros

Fuente: Elaboración propia

A partir de estos datos extraídos, se toman 03 muestras de tamaño 99 cada una.

Para validar la no redundancia de datos se utilizó la moda como estadístico y se obtuvieron los resultados que confirman que no existen códigos repetidos.

```
Estadísticas descriptivas: muestra_1; muestra_2; muestra_3

                    N para
Variable   Modo    moda
muestra_1     *       0
muestra_2     *       0
muestra_3     *       0
```

Figura 15. No redundancia de datos del campo código

Fuente: Elaboración propia

Se llama a los procedimientos de búsqueda de ambas estructuras de datos dinámicas y se almacenan los tiempos de respuesta y la cantidad de comparaciones realizadas para localizar dichos registros. Los resultados obtenidos son:

↓	C1	C2	C3	C4	C5	C6	C7	C8
	c_avl_c	c_rb_c		c_avl_d	c_rb_d		c_avl_a	c_rb_a
1	9	10		10	9		11	12
2	13	14		8	7		10	12
3	13	12		12	14		12	8
4	12	11		11	10		11	10
5	13	12		8	10		11	12
6	10	11		12	11		13	12
7	10	11		13	13		12	13
8	10	11		13	14		13	12
9	12	13		8	7		12	11
10	13	12		7	8		12	10
11	11	10		13	12		11	12
12	13	15		13	13		12	12
13	13	14		13	13		13	13
14	12	13		13	13		14	13
15	12	13		13	14		11	13
16	12	12		13	12		13	13
17	13	12		12	11		11	12
18	13	12		11	14		12	12

Figura 16. Datos correspondientes a tres muestras de 99 registros para el número de comparaciones en la búsqueda de datos.

Fuente: Elaboración propia

Aplicando el software Minitab se efectuaron los cálculos estadísticos para las 03 muestras, obteniéndose los siguientes resultados:

Estadísticas descriptivas: c_avl_c; c_rb_c; c_avl_d; c_rb_d; c_avl_a; c_rb_a

Variable	Media	Desv.Est.	CoefVar	Mediana	Modo	N para moda
c_avl_c	11,424	1,709	14,96	12,000	13	32
c_rb_c	11,525	1,955	16,97	12,000	12	31
c_avl_d	11,848	1,554	13,12	12,000	13	43
c_rb_d	12,000	2,104	17,54	12,000	12	26
c_avl_a	11,960	1,505	12,58	12,000	12	29
c_rb_a	12,071	1,473	12,20	12,000	12; 13	28

Figura 17. Estadísticas descriptivas para las comparaciones en las 03 muestras

Fuente: Elaboración propia

Interpretación de resultados:

Datos ingresados en orden creciente (c): Para esta primera muestra se observa que el número de comparaciones promedio para localizar un registro almacenado en un árbol AVL (11,424 comparaciones) es menor que el número de comparaciones promedio necesarios para localizarlo en un árbol *Red–Black* (11,525 comparaciones).

Datos ingresados en orden decreciente (d): Para la segunda muestra se observa que el número de comparaciones promedio para localizar un registro almacenado en un árbol AVL (11,848 comparaciones) es menor que el número de comparaciones promedio necesarios para localizarlo en un árbol *Red–Black* (12 comparaciones).

Datos ingresados en orden aleatorio (a): Para la tercera muestra se observa que el número de comparaciones promedio para localizar un registro almacenado en un árbol AVL (11,960 comparaciones) es menor que el número de comparaciones promedio necesarios para localizarlo en un árbol *Red–Black* (12,071 comparaciones).

Respecto a la Desviación Estándar de los datos:

Datos ingresados en orden creciente (c): En promedio, los datos se alejan 1,709 unidades respecto a la media del número de comparaciones para la búsqueda en árboles AVL. Así mismo, en promedio, los datos se

alejan 1,955 unidades respecto a la media del número de comparaciones para la búsqueda en árboles *Red–Black*.

Datos ingresados en orden decreciente (d): En promedio, los datos se alejan 1,554 unidades respecto a la media del número de comparaciones para la búsqueda en árboles AVL. Así mismo, en promedio, los datos se alejan 2,104 unidades respecto a la media del número de comparaciones para la búsqueda en árboles *Red–Black*.

Datos ingresados en orden aleatorio (a): En promedio, los datos se alejan 1,505 unidades respecto a la media del número de comparaciones para la búsqueda en árboles AVL. Así mismo, en promedio, los datos se alejan 1,473 unidades respecto a la media del número de comparaciones para la búsqueda en árboles *Red–Black*.

Del estadígrafo Coeficiente de Variación, se observa que los datos se encuentran más dispersos en los árboles *Red–Black* con respecto a los árboles AVL, esta diferencia se acentúa más en las dos primeras muestras (provenientes de los registros ingresados en orden creciente y en orden decreciente). Esto se explica por la forma en cómo se estructuran los nodos en el árbol *Red–Black*, un árbol *Red–Black* tiene un peor caso de equilibrio cuando los datos son ingresados en orden creciente o decreciente.

Respecto a la inserción de los datos:

Se obtuvo los siguientes tiempos:

Datos ingresados en orden creciente:

```
Tiempo de inserción en el árbol AVL: 7 935,92 ms
Altura del árbol AVL: 13
Tiempo de inserción en el árbol Red-Black: 5 204,01 ms
Altura del árbol RB: 23
```

Datos ingresados en orden decreciente:

```
Tiempo de inserción en el árbol AVL: 6 650,85 ms
Altura del árbol AVL: 13
Tiempo de inserción en el árbol Red-Black: 5 849,91 ms
Altura del árbol RB: 23
```

Datos ingresados en orden aleatorio:

```
Tiempo de inserción en el árbol AVL: 7 673,32 ms
Altura del árbol AVL: 15
Tiempo de inserción en el árbol Red-Black: 5 223,57 ms
Altura del árbol RB: 16
```

Se puede observar que en los tres casos, el tiempo de inserción de datos en un árbol AVL es superior que el tiempo de inserción en un árbol *Red–Black* para los 6 966 registros.

Respecto a la altura del árbol, se observó que el árbol *Red–Black* se estructura de forma más equilibrada cuando los datos son insertados en

forma aleatoria (sin un orden específico), mientras que tiene su peor equilibrio (mayor altura) cuando los datos son insertados en orden creciente como decreciente.

CAPÍTULO V

DISCUSIÓN

Se aplicó la prueba de comparación de medias para los datos de las 03 muestras de los tiempos de respuesta en el proceso de búsqueda de datos contenidos en un árbol AVL y un árbol *Red–Black*. Se utilizó el software Minitab v16 y se obtuvo los siguientes resultados:

Muestra 1 (Datos ingresados en orden creciente):

H_o: El tiempo de respuesta promedio para la búsqueda de registros localizados en un árbol AVL es igual al tiempo de respuesta promedio para la búsqueda de registros localizados en un árbol *Red–Black*.

$$H_o\text{: } \mu_{t_AVL} = \mu_{t_RB}$$

H_a: El tiempo de respuesta promedio para la búsqueda de registros localizados en un árbol AVL es diferente al tiempo de respuesta promedio para la búsqueda de registros localizados en un árbol *Red–Black*.

$$H_a\text{: } \mu_{t_AVL} \neq \mu_{t_RB}$$

Aplicando la prueba t para 2 muestras, con un nivel de significancia del α = 5 % y un nivel de confianza del 95 %, se obtiene:

```
Prueba T e IC de dos muestras: t_avl_c; t_rb_c

T de dos muestras para t_avl_c vs. t_rb_c

                                   Error
                                estándar
                                   de la
          N  Media  Desv.Est.      media
t_avl_c  99   5,34       1,63       0,16
t_rb_c   99   5,09       1,58       0,16

Diferencia = mu (t_avl_c) - mu (t_rb_c)
Estimado de la diferencia:  0,254
IC de 95 % para la diferencia:  (-0,196; 0,704)
Prueba T de diferencia = 0 (vs. no =): Valor T = 1,11  Valor P = 0,267
GL =
     196
Ambos utilizan Desv.Est. agrupada = 1,6052
```

Figura 18. Prueba T para los tiempos de respuesta en la muestra 1

Fuente: Elaboración propia

Al ser el valor de p = 0,267 mayor que nuestro α = 0,050 se acepta H_o y se rechaza H_a. Es decir, el tiempo de respuesta promedio para la búsqueda de registros localizados en un árbol AVL es equivalente al tiempo promedio de respuesta para la búsqueda de registros localizados en un árbol *Red–Black*.

Muestra 2 (Datos ingresados en orden decreciente):

H_o: El tiempo de respuesta promedio para la búsqueda de registros localizados en un árbol AVL es igual al tiempo de respuesta promedio para la búsqueda de registros localizados en un árbol *Red–Black*.

$$H_o:\ \mu_{t_AVL} = \mu_{t_RB}$$

H_a: El tiempo de respuesta promedio para la búsqueda de registros localizados en un árbol AVL es diferente al tiempo de respuesta promedio para la búsqueda de registros localizados en un árbol *Red–Black*.

$$H_a:\ \mu_{t_AVL} \neq \mu_{t_RB}$$

Aplicando la prueba t para 2 muestras, con un nivel de significancia del α = 5% y un nivel de confianza del 95 %, se obtiene:

```
Prueba T e IC de dos muestras:

T de dos muestras para t_avl_d vs. t_rb_d

                                  Error estándar
          N  Media  Desv.Est.     de la media
t_avl_d  99   6,96       4,25      0,43
t_rb_d   99   6,48       4,34      0,44

Diferencia = mu (t_avl_d) - mu (t_rb_d)
Estimado de la diferencia:  0,477
IC de 95 % para la diferencia:  (-0,728; 1,682)
Prueba T de diferencia = 0 (vs. no =): Valor T = 0,78  Valor P = 0,436
```

Figura 19. Prueba T para los tiempos de respuesta en la muestra 2

Fuente: Elaboración propia

Al ser el valor de p = 0,436 mayor que nuestro α = 0,050 se acepta H_o y se rechaza H_a. Es decir, el tiempo de respuesta promedio para la búsqueda de registros localizados en un árbol AVL es equivalente al tiempo promedio de respuesta para la búsqueda de registros localizados en un árbol *Red–Black*.

Muestra 3 (Datos ingresados en orden aleatorio):

H_o: El tiempo de respuesta promedio para la búsqueda de registros localizados en un árbol AVL es igual al tiempo de respuesta promedio para la búsqueda de registros localizados en un árbol *Red–Black*.

$$H_o:\ \mu_{t_AVL} = \mu_{t_RB}$$

H_a: El tiempo de respuesta promedio para la búsqueda de registros localizados en un árbol AVL es diferente al tiempo de respuesta promedio para la búsqueda de registros localizados en un árbol *Red–Black*.

$$H_a:\ \mu_{t_AVL} \neq \mu_{t_RB}$$

Aplicando la prueba t para 2 muestras, con un nivel de significancia del α = 5 % y un nivel de confianza del 95 %, se obtiene:

```
Prueba T e IC de dos muestras: t_avl_a; t_rb_a

T de dos muestras para t_avl_a vs. t_rb_a

                                  Error
                               estándar
                                  de la
          N  Media  Desv.Est.     media
t_avl_a  99   7,93       6,07      0,61
t_rb_a   99   7,87       6,61      0,66


Diferencia = mu (t_avl_a) - mu (t_rb_a)
Estimado de la diferencia:  0,063
IC de 95 % para la diferencia:  (-1,716; 1,842)
Prueba T de diferencia = 0 (vs. no =): Valor T = 0,07  Valor P = 0,944
GL =
     196
Ambos utilizan Desv.Est. agrupada = 6,3480
```

Figura 20. Prueba T para los tiempos de respuesta en la muestra 3
Fuente: Elaboración propia

Al ser el valor de p = 0,944 mayor que nuestro α = 0,050 se acepta H_o y se rechaza H_a. Es decir, el tiempo de respuesta promedio para la búsqueda de registros localizados en un árbol AVL es igual al tiempo promedio de respuesta para la búsqueda de registros localizados en un árbol *Red–Black*.

Por los resultados observados en las tres muestras anteriores se concluye que el tiempo de respuesta promedio para la búsqueda de registros localizados en un árbol AVL es equivalente al tiempo promedio de respuesta para la búsqueda de registros localizados en un árbol *Red–Black*.

Respecto al ***número de comparaciones*** necesarias para localizar una clave dentro de las dos estructuras, se plantearon las siguientes hipótesis estadísticas:

H_o: El número de comparaciones en un árbol AVL es igual al número de comparaciones en un árbol *Red–Black*, para localizar un dato.

$$H_o:\ \mu_{c_AVL} = \mu_{c_RB}$$

H_a: El número de comparaciones en un árbol AVL es diferente al número de comparaciones en un árbol *Red–Black*, para localizar un dato.

$$H_a:\ \mu_{c_AVL} \neq \mu_{c_RB}$$

Aplicando la prueba t para 2 muestras, con un nivel de significancia del α = 5 % y un nivel de confianza del 95 %, se obtiene:

```
Prueba T e IC de dos muestras: c_avl_a; c_rb_a

T de dos muestras para c_avl_a vs. c_rb_a

                                  Error estándar
         N  Media  Desv.Est.      de la media
c_avl_a  99  11,96       1,50       0,15
c_rb_a   99  12,07       1,47       0,15

Diferencia = mu (c_avl_a) - mu (c_rb_a)
Estimado de la diferencia:  -0,111
IC de 95 % para la diferencia:  (-0,528; 0,306)
Prueba T de diferencia = 0 (vs. no =): Valor T = -0,53  Valor P =
0,600  GL = 195
```

Figura 21. Prueba T para las comparaciones

Fuente: Elaboración propia

Al ser el valor de $p = 0{,}600$ mayor que nuestro $\alpha = 0{,}050$ se acepta H_o y se rechaza H_a. Es decir, el número de comparaciones en un árbol AVL es equivalente al número de comparaciones en un árbol *Red–Black*, para localizar un dato.

CONCLUSIONES

1. Las velocidades de respuesta en la búsqueda de datos contenidos en estructuras de datos dinámicas del tipo árbol AVL y árbol *Red–Black* son estadísticamente equivalentes, pero se observó también que la altura de los árboles son diferentes debido a la forma en cómo han sido insertados los datos (en orden ascendente, descendente o aleatorio), esto conlleva a que exista una diferencia en los tiempos de inserción de los datos; el tiempo necesario para insertar datos en un árbol AVL es mayor que el tiempo necesario para insertar los mismos datos en un árbol *Red–Black*: 7 420 milisegundos y 5 425,83 milisegundos, respectivamente.

2. El tiempo que se tarda en localizar un dato contenido en una estructura de datos dinámica del tipo árbol AVL es equivalente al tiempo que se tarda en localizar el mismo dato contenido en una estructura dinámica del tipo árbol *Red–Black*: 7,93 milisegundos y 7,87 milisegundos, respectivamente.

3. El número de comparaciones necesarias para localizar un dato contenido en una estructura de datos dinámica del tipo árbol AVL es equivalente al número de comparaciones necesarias para localizar el

mismo dato en una estructura de datos dinámica del tipo árbol *Red–Black*. En promedio 11,96 comparaciones y 12,07 comparaciones respectivamente.

RECOMENDACIONES

1. Se recomienda continuar la presente investigación para las operaciones de inserción, modificación y eliminación de datos. Además de ampliar las estructuras aquí analizadas, como por ejemplo con los árboles *Splay*, árboles B, B+, grafos dirigidos, entre otros, para verificar los tiempos de respuesta de los procesos de búsqueda para estas estructuras.

2. Aumentar el número de investigaciones orientadas a desarrollar las Ciencias de la Computación para que a partir del entendimiento de las bases teóricas se puedan crear nuevas teorías que las enriquezcan.

REFERENCIAS BIBLIOGRÁFICAS

Andersson, A. (1993). Balanced Search Trees Made Simple. *Workshop on Algorithms and Data Structures*, 60-71.

Bowman, C. F. (1999). *Algoritmos y estructura de datos.* Mexico: Oxford.

Bronson, N., Casper, J., Chafi, H., & Olukotun, K. (2010). A Practical Concurrent Binary Search Tree. *ACM Sigplan*, 257-268 .

Caicedo, A., Wagner, G., & Méndez, R. (2010). *Introducción a la Teoría de Grafos.* Quindió: Elizcom.

Cairó, O., & Guardati, S. (2002). *Estructura de datos.* Mexico: Mc Graw Hill.

Cormen, T., Leiserson, C., Rivest, R., & Stein, C. (2009). *Introduction to Algorithms.* London: The MIT Press.

Cortez, A., & Vega, H. (s.f.). Árboles Biselados. *Revista de Ingeniería de Sistemas e Informática*, 39-49.

Dominique, H. (2004). A Disquisition on The Performance Behaviour of Binary Search Tree Data Structures. *Upgrade CEPIS*, 67-75.

Drozdek, A. (2007). *Estructura de datos y algoritmos con Java.* México: Cigange Editores.

Flores, R. (2005). *Algoritmos, estructuras de datos y programación orientada a objetos.* Bogotá: Ecoe Ediciones.

Forouzan, B., & Chung, S. (2003). *Foundations of Computer Science: From*

Data Manipulation to Theory of Computation. Cengage Learning.

Franch, X. (1999). *Estructuras de datos: Especificación, diseño e implementación.* Univ. Politec. de Cataluña, 1999.

Gomez, A., & Ania, I. (2008). *Introducción a la Computación.* Mexico DF: Cengage Learning.

Hinojosa, E. (2014). *Velocidad de respuesta de los algoritmos de búsqueda de datos contenidos en estructuras de datos estáticas y dinámicas* (Tesis de maestria). Tacna: Universidad Nacional Jorge Basadre Grohmann.

Joyanes, L. (2002). *Programación en C: Metodología, algoritmos y estructura de datos.* Madrid: Mc Graw Hill.

Joyanes, L. (2008). *Fundamentos de programación.* Madrid: McGraw Hill.

Karlton, P. (1976). Performance of height-balanced trees. *Communications of the ACM*, 23-28.

Knuth, D. (2000). *Sorting and searching.* Boston: Addison-Wesley.

Krishnamoorthy, R. (2008). *Data Structures Using C.* New Delhi: McGraw.

Langsam, Y. (1996). *Data Structure using C and C++.* Prentice Hall.

Loomis, M. (1999). *Estructura de datos y organización de archivos.* Prentice Hall.

Martinez, R., & Quiroga, E. (2002). *Estructura de datos: referencia práctica con orientación a objetos.* Cengage Learning.

Murillo, J., & Caamaño, S. (2013). Comparación entre algoritmos recursivos e iterativos y su medición en términos de eficiencia. *Uniciencia*, 341.

Pfaff, B. (2004). Performance Analysis of BSTs in System Software. *Sigmetrics*, 410-411.

Puntambekar, A. (2009). *Data Structures And Algorithms.* Pune: Technical Publications.

S.I. Didact. (2005). *Manual de programacion lenguaje C++.* España: MAD-Eduforma.

Silva, L. (11 de Enero de 2017). Obtenido de Árboles coloreados. Red black: http://www2.elo.utfsm.cl/~lsb/elo320/clases/c12.pdf

Tanenbaum, A., & Augenstein, M. (1993). *Estructuras de Datos en C.* New York: Prentice Hall.

Urdaneta, G. (25 de Febrero de 2010). *Árboles Rojo-Negros.* Recuperado el 14 de Marzo de 2013, de Universidad de Zulia: http://www.ica.luz.edu.ve/eda/guias/rojonegros.pdf

Weiss, M. (2000). *Estructuras de datos en Java.* México: Pearson Educación.

Wirth, N. (1994). *Algoritmos + Estructura de Datos.* México: Prentice Hall.

Ziviani, N. (2007). *Diseño de algoritmos con implementaciones en Pascal y C.* Madrid: Paraninfo.

ANEXOS

ANEXO 1: Búsquedas y tiempo de respuesta de 99 registros

MUESTRA 1			
BÚSQUEDA EN DATOS INSERTADOS EN ORDEN CRECIENTE			
N°	CÓDIGO	T_AVL_C	T_RB_C
1	2015111019	5.02857	4.74921
2	2016101015	4.19048	4.74921
3	201339404	4.46984	3.91111
4	2011130011	3.91111	3.91111
5	2011124002	3.63175	3.91111
6	2015112023	4.19048	4.19048
7	2015105055	3.35238	3.07302
8	2015101002	3.91111	3.91111
9	2015109006	5.30794	4.46984
10	201236749	5.30794	5.5873
11	201237052	4.19048	3.63175
12	2016119052	6.14603	6.98413
13	2016102071	5.02857	5.5873
14	2014127007	4.19048	4.74921
15	2015125020	5.30794	4.74921
16	200933971	3.91111	4.19048
17	200528047	5.86667	3.91111
18	201339261	5.86667	5.30794
19	2014124025	7.54286	8.66032
20	201338822	4.74921	4.19048
21	2011107010	6.14603	5.5873
22	200832257	5.02857	4.19048
23	2015102017	5.5873	4.19048
24	2014129034	4.19048	4.46984
25	201339344	6.70476	4.19048
26	201237072	3.91111	4.74921
27	2014126048	3.91111	4.19048
28	201339204	5.86667	4.74921
29	2015178009	3.91111	3.91111
30	201237565	5.86667	3.91111
31	2016129005	7.26349	9.77778
32	200934010	4.19048	4.74921
33	2011122048	5.5873	4.19048
34	200730932	3.91111	4.19048
35	2015110024	3.35238	3.35238
36	201236311	9.49841	8.38095
37	2014102074	4.74921	4.19048
38	2016111007	6.14603	5.02857
39	2014129030	8.10159	5.86667
40	2016102015	4.19048	5.02857
41	201034807	6.14603	5.30794
42	2015108036	5.02857	5.30794
43	2011119053	12.5714	6.70476
44	2014102003	4.74921	4.46984
45	201236686	3.63175	4.46984
46	2011118006	6.14603	6.14603
47	2015178018	4.19048	4.46984
48	2011119036	9.49841	6.4254
49	201338472	5.86667	5.30794
50	201236238	6.14603	5.5873
51	2015106009	8.10159	6.4254
52	2015101019	4.19048	4.19048
53	2014119015	3.91111	3.91111
54	200933989	6.98413	4.19048
55	201339512	4.74921	4.74921
56	200629923	5.02857	5.5873
57	2014118011	4.46984	4.19048
58	201339272	5.02857	3.91111
59	201236664	6.14603	5.86667
60	200832664	4.19048	4.19048
61	2016109031	6.14603	5.5873
62	2016105001	3.35238	5.02857
63	2015114014	3.35238	3.91111
64	201338912	5.02857	3.91111
65	200731124	5.02857	4.46984
66	201338563	7.26349	6.4254
67	2016131019	5.5873	8.38095
68	2015106043	4.19048	4.19048
69	2015111020	9.77778	13.9683
70	200933384	4.19048	5.02857
71	2016105041	4.46984	5.02857
72	201338960	5.5873	5.02857
73	2011122005	4.19048	3.91111
74	200832075	6.14603	7.54286
75	201338223	4.46984	4.46984
76	200323871	5.30794	4.46984
77	201338882	4.19048	5.30794
78	201338108	6.14603	4.19048
79	200934041	6.70476	6.4254
80	2011114049	4.46984	4.19048
81	2016119014	6.98413	4.74921
82	2011109041	10.8952	10.3365
83	2014178002	4.46984	3.91111
84	201237557	5.5873	4.19048
85	2014102041	6.98413	3.91111
86	2014115002	4.46984	4.74921
87	2011127014	4.19048	5.30794
88	2015110041	4.46984	6.70476
89	2014125011	3.91111	4.19048
90	2014118008	4.74921	4.46984
91	201339523	6.98413	6.4254
92	2011126042	5.86667	4.46984
93	2015119009	4.46984	4.19048
94	2014102084	5.5873	4.74921
95	201035440	4.46984	4.74921
96	201338743	4.46984	3.91111
97	200933556	3.91111	6.14603
98	2014130041	4.46984	4.74921
99	201236155	5.30794	5.5873

MUESTRA 2			
BÚSQUEDA EN DATOS INSERTADOS EN ORDEN DECRECIENTE			
N°	CÓDIGO	T_AVL_D	T_RB_D
1	2015111056	6.4254	4.46984
2	2016121013	5.02857	3.35238
3	201236809	4.46984	3.91111
4	2016111023	3.91111	3.91111
5	201236795	3.35238	3.63175
6	2016123018	11.1746	7.82222
7	201339174	7.26349	5.5873
8	201034701	7.82222	7.82222
9	2015118048	4.19048	3.63175
10	201338073	3.63175	3.63175
11	2016104038	5.02857	4.46984
12	2014118015	6.98413	5.86667
13	2014112033	4.19048	3.91111
14	2014119021	4.74921	4.46984
15	201338332	5.30794	5.02857
16	2016110016	4.74921	4.19048
17	2016179025	6.14603	5.86667
18	201236276	4.19048	5.5873
19	201338406	4.46984	4.19048
20	2015126031	5.30794	4.46984
21	199917478	4.19048	5.02857
22	2014102027	7.26349	5.5873
23	2011129034	4.46984	7.26349
24	2016117024	5.02857	4.74921
25	201237589	5.30794	5.5873
26	2015104016	7.82222	5.02857
27	2016124027	4.74921	4.19048
28	2016179014	8.38095	5.86667
29	2015110032	4.19048	4.74921
30	199917454	4.19048	5.30794
31	2016129025	9.49841	9.21905
32	2016101034	5.02857	4.46984
33	201339444	4.74921	4.74921
34	2011114049	7.54286	5.02857
35	2016126023	4.19048	4.19048
36	2014111008	6.98413	5.30794
37	2015110006	4.46984	3.91111
38	2015106054	7.26349	5.5873
39	200528136	4.74921	6.14603
40	2016110017	8.38095	5.02857
41	2016127030	6.98413	6.70476
42	2016122009	4.19048	3.63175
43	201035673	9.77778	13.4095
44	2016112003	5.02857	4.74921
45	2016101006	3.91111	3.91111
46	201339149	6.14603	6.4254
47	2016113035	4.19048	3.35238
48	2011128039	6.70476	7.26349
49	2016129033	4.74921	4.19048
50	2011127030	5.30794	5.02857
51	2016115015	6.4254	6.4254
52	201236070	4.46984	4.74921
53	2015122013	7.26349	6.14603
54	201338749	29.3333	36.0381
55	2015103052	8.38095	6.4254
56	201338348	4.74921	5.02857
57	201237455	8.38095	12.8508
58	201339213	11.1746	8.38095
59	2011118029	7.26349	8.38095
60	2016128038	6.70476	4.74921
61	2016107050	7.26349	9.21905
62	2015118040	6.98413	6.14603
63	2011126008	11.1746	5.5873
64	2014123020	6.4254	10.6159
65	2014108002	6.98413	8.93968
66	2014131002	5.02857	4.46984
67	2014129008	6.4254	6.14603
68	201338792	8.38095	7.26349
69	201237438	10.6159	6.14603
70	2014105002	10.0571	6.70476
71	2011105002	4.46984	5.02857
72	2014110008	5.30794	14.2476
73	2016106065	8.38095	4.74921
74	2014104009	4.74921	4.46984
75	201034859	5.86667	5.02857
76	2016127007	7.26349	6.70476
77	2015102032	21.2317	13.1302
78	2016130032	8.38095	7.26349
79	201339390	7.26349	10.8952
80	2014122014	8.66032	7.26349
81	2015131007	4.74921	3.91111
82	2016112044	9.77778	6.4254
83	2014120028	6.70476	6.14603
84	2016118016	29.6127	26.5397
85	2016110001	18.7175	15.0857
86	200933422	3.63175	3.63175
87	2016128045	9.77778	4.46984
88	201338313	5.02857	4.74921
89	201236124	5.02857	6.70476
90	2016122020	8.66032	6.4254
91	2015110036	4.74921	6.70476
92	2016128049	6.70476	5.86667
93	2015129032	4.46984	3.91111
94	2016109022	5.02857	5.86667
95	201034806	3.35238	3.63175
96	2015119029	4.74921	4.74921
97	2015131015	6.4254	5.30794
98	2016126015	5.02857	5.02857
99	2014106049	7.54286	5.5873

MUESTRA 3			
BÚSQUEDA EN DATOS INSERTADOS EN ORDEN ALEATORIO			
N°	CÓDIGO	T_AVL_A	T_RB_A
1	2016178038	5.5873	5.30794
2	2015108045	4.19048	3.91111
3	201338325	4.19048	3.35238
4	2016128060	3.91111	3.91111
5	201338033	3.91111	3.91111
6	2014103037	14.527	4.19048
7	2014111029	3.91111	4.46984
8	2015113005	5.30794	4.46984
9	2016109014	4.46984	4.46984
10	2016113039	5.30794	5.86667

11	2014111016	5.5873	5.5873
12	2011122015	4.46984	4.46984
13	2016111013	21.7905	16.2032
14	2015108027	5.30794	6.98413
15	2015104018	4.46984	4.19048
16	200933825	6.4254	6.14603
17	2011128021	5.30794	6.14603
18	2015103031	10.0571	5.86667
19	2016111026	6.4254	4.74921
20	2015112006	4.46984	3.91111
21	2014128054	5.02857	5.30794
22	201236015	6.98413	15.0857
23	2011123024	5.86667	5.86667
24	2016101005	5.30794	4.46984
25	2011101045	6.70476	5.02857
26	201236306	4.74921	7.54286
27	2016111028	10.0571	8.38095
28	2011119030	4.46984	4.46984
29	2015126028	9.77778	7.26349
30	2014110008	3.63175	4.46984
31	2016109043	10.6159	9.49841
32	2011110029	8.10159	7.54286
33	2015117006	5.30794	4.46984
34	2014105080	4.46984	4.74921
35	201339156	3.91111	5.30794
36	2011127033	3.63175	3.91111
37	2015119025	5.86667	6.14603
38	201339098	6.14603	6.70476
39	2015118055	4.19048	8.46984
40	2014111008	6.14603	6.70476
41	2015127002	4.74921	3.91111
42	2016126016	4.74921	4.74921
43	201237617	4.46984	5.86667
44	201338079	5.02857	5.02857
45	201236357	8.93968	7.26349
46	201035536	11.454	11.7333
47	201339164	6.70476	7.26349
48	2015118037	7.26349	5.30794
49	201338211	3.63175	3.63175
50	201236345	8.66032	13.4095
51	201338371	4.19048	5.30794
52	2016102015	4.74921	4.46984
53	201237076	5.5873	7.54286
54	2015108005	5.5873	6.4254
55	200934232	5.86667	5.86667
56	2016102003	5.30794	5.5873
57	2015131003	5.5873	5.86667
58	2011128003	6.98413	5.02857
59	2014101032	5.5873	8.38095
60	2014104032	6.14603	5.02857
61	201338370	8.38095	10.6159
62	2014127018	9.49841	10.3365
63	2014119014	10.0571	10.3365
64	2015177005	12.2921	10.0571
65	2015106081	8.93968	9.21905
66	201338339	12.5714	11.7333
67	2016178008	8.38095	12.0127
68	2016111035	8.93968	8.38095
69	201236085	7.26349	10.0571
70	2014123008	14.2476	13.9683
71	201338064	9.21905	13.4095
72	201236263	27.0984	23.746
73	2011126043	13.9683	13.4095
74	201237052	55.4857	64.5333
75	2014129030	16.7619	12.0127
76	2014112012	6.98413	6.70476
77	2014113004	11.1746	6.98413
78	201236806	4.19048	6.14603
79	2015105015	6.14603	6.4254
80	201236286	6.4254	5.5873
81	2016105052	9.21905	5.86667
82	201237645	10.3365	7.26349
83	2016120042	10.6159	6.70476
84	200934098	10.6159	7.54286
85	200934473	9.21905	8.66032
86	201035114	7.82222	8.93968
87	2015129031	6.70476	9.21905
88	2015101026	8.10159	8.10159
89	2015102052	6.70476	6.70476
90	201338185	8.38095	11.1746
91	2014127036	7.26349	6.98413
92	2016110054	7.54286	6.4254
93	2016102075	6.70476	6.4254
94	2016113038	5.86667	5.30794
95	201035639	8.66032	9.21905
96	2016114014	5.86667	5.30794
97	200527873	7.54286	9.49841
98	200832491	6.98413	6.4254
99	2016128009	10.8952	10.0571

ANEXO 2: Reporte de número de comparaciones para localizar datos.

MUESTRA 1			
BÚSQUEDA EN DATOS INSERTADOS EN ORDEN CRECIENTE			
N°	CÓDIGO	C_AVL_C	C_RB_C
1	2015111019	9	10
2	2016101015	13	14
3	201339404	13	12
4	2011130011	12	11
5	2011124002	13	12
6	2015112023	10	11
7	2015105055	10	11
8	2015101002	10	11
9	2015109006	12	13
10	201236749	13	12
11	201237052	11	10
12	2016119052	13	15
13	2016102071	13	14
14	2014127007	12	13
15	2015125020	12	13
16	200933971	12	12
17	200528047	13	12
18	201339261	13	12
19	2014124025	10	11
20	201338822	12	11
21	2011107010	12	12
22	200832257	10	11
23	2015102017	11	12
24	2014129034	12	13
25	201339344	13	12
26	201237072	13	12
27	2014126048	11	12
28	201339204	12	11
29	2015178009	8	9
30	201237565	10	9
31	2016129005	12	16
32	200934010	11	11
33	2011122048	12	11
34	200730932	13	12
35	2015110024	4	6
36	201236311	13	12
37	2014102074	11	10
38	2016111007	12	14
39	2014129030	11	12
40	2016102015	12	13
41	201034807	9	9
42	2015108036	12	13
43	2011119053	13	12
44	2014102003	11	10
45	201236686	7	6
46	2011118006	10	9
47	2015178018	10	11
48	2011119036	13	12
49	201338472	11	10
50	201236238	13	12
51	2015106009	10	11
52	2015101019	12	13
53	2014119015	12	12
54	200933989	8	8
55	201339512	13	12
56	200629923	12	11
57	2014118011	13	13
58	201339272	12	11
59	201236664	13	12
60	200832664	11	12
61	2016109031	13	15
62	2016105001	13	15
63	2015114014	10	11
64	201338912	12	11
65	200731124	13	12
66	201338563	13	12
67	2016131019	12	17
68	2015106043	9	10
69	2015111020	12	13
70	200933384	11	12
71	2016105041	13	15
72	201338960	9	8
73	2011122005	13	12
74	200832075	13	12
75	201338223	13	12
76	200323871	11	10
77	201338882	11	10
78	201338108	10	9
79	200934041	11	11
80	2011114049	13	12
81	2016119014	9	11
82	2011109041	13	12
83	2014178002	9	10
84	201237557	7	6
85	2014102041	13	12
86	2014115002	13	13
87	2011127014	9	8
88	2015110041	12	13
89	2014125011	12	13
90	2014118008	13	13
91	201339523	12	11
92	2011126042	8	7
93	2015119009	12	13
94	2014102084	13	12
95	201035440	11	13
96	201338743	12	11
97	200933556	12	13
98	2014130041	12	13
99	201236155	10	9

MUESTRA 2			
BÚSQUEDA EN DATOS INSERTADOS EN ORDEN DECRECIENTE			
Nº	CÓDIGO	C_AVL_D	C_RB_D
1	2015111056	10	9
2	2016121013	8	7
3	201236809	12	14
4	2016111023	11	10
5	201236795	8	10
6	2016123018	12	11
7	201339174	13	13
8	201034701	13	14
9	2015118048	8	7
10	201338073	7	8
11	2016104038	13	12
12	2014118015	13	13
13	2014112033	13	13
14	2014119021	13	13
15	201338332	13	14
16	2016110016	13	12
17	2016179025	12	11
18	201236276	11	14
19	201338406	13	14
20	2015126031	7	6
21	199917478	12	18
22	2014102027	13	13
23	2011129034	11	15
24	2016117024	12	11
25	201237589	12	13
26	2015104016	11	10
27	2016124027	12	11
28	2016179014	13	12
29	2015110032	12	11
30	199917454	11	17
31	2016129025	13	12
32	2016101034	13	12
33	201339444	13	13
34	2011114049	12	9
35	2016126023	11	10
36	2014111008	12	12
37	2015110006	11	10
38	2015106054	12	11
39	200528136	11	16
40	2016110017	11	10
41	2016127030	12	11
42	2016122009	9	8
43	201035673	13	14
44	2016112003	13	12
45	2016101006	13	12
46	201339149	13	13
47	2016113035	11	10
48	2011128039	11	15
49	2016129033	13	12
50	2011127030	11	15
51	2016115015	12	11
52	201236070	12	15
53	2015122013	12	11
54	201338749	12	13
55	2015103052	13	12
56	201338348	10	11
57	201237455	12	13
58	201339213	13	13
59	2011118029	12	14
60	2016128038	12	11
61	2016107050	12	11
62	2015118040	12	11
63	2011126008	12	15
64	2014123020	12	12
65	2014108002	13	13
66	2014131002	13	12
67	2014129008	12	11
68	201338792	13	14
69	201237438	12	13
70	2014105002	13	13
71	2011105002	10	16
72	2014110008	13	13
73	2016106065	13	12
74	2014104009	11	11
75	201034859	13	14
76	2016127007	13	12
77	2015102032	12	11
78	2016130032	11	10
79	201339390	13	13
80	2014122014	13	13
81	2015131007	13	12
82	2016112044	13	12
83	2014120028	10	10
84	2016118016	13	12
85	2016110001	11	10
86	200933422	9	11
87	2016128045	13	12
88	201338313	13	14
89	201236124	12	15
90	2016122020	13	12
91	2015110036	13	12
92	2016128049	13	12
93	2015129032	13	12
94	2016109022	13	12
95	201034806	5	6
96	2015119029	13	12
97	2015131015	12	11
98	2016126015	13	12
99	2014106049	12	12

MUESTRA 3			
BÚSQUEDA EN DATOS INSERTADOS EN ORDEN ALEATORIO			
Nº	CÓDIGO	C_AVL_A	C_RB_A
1	2016178038	11	12
2	2015108045	10	12
3	201338325	12	8
4	2016128060	11	10
5	201338033	11	12
6	2014103037	13	12
7	2014111029	12	13
8	2015113005	13	12

9	2016109014	12	11
10	2016113039	12	10
11	2014111016	11	12
12	2011122015	12	12
13	2016111013	13	13
14	2015108027	14	13
15	2015104018	11	13
16	200933825	13	13
17	2011128021	11	12
18	2015103031	12	12
19	2016111026	12	12
20	2015112006	11	11
21	2014128054	13	14
22	201236015	12	13
23	2011123024	9	10
24	2016101005	13	12
25	2011101045	12	12
26	201236306	12	11
27	2016111028	14	11
28	2011119030	12	12
29	2015126028	13	14
30	2014110008	13	13
31	2016109043	10	11
32	2011110029	12	11
33	2015117006	14	12
34	2014105080	11	14
35	201339156	12	9
36	2011127033	8	9
37	2015119025	12	11
38	201339098	13	12
39	2015118055	5	10
40	2014111008	11	13
41	2015127002	14	13
42	2016126016	13	13
43	201237617	11	14
44	201338079	13	15
45	201236357	11	13
46	201035536	13	13
47	201339164	12	12
48	2015118037	12	9
49	201338211	10	10
50	201236345	11	12
51	201338371	11	12
52	2016102015	13	13
53	201237076	11	12
54	2015108005	10	11
55	200934232	12	12
56	2016102003	12	10
57	2015131003	13	10
58	2011128003	12	13
59	2014101032	13	13
60	2014104032	11	10
61	201338370	12	11
62	2014127018	14	15
63	2014119014	9	9
64	2015177005	12	13
65	2015106081	13	14
66	201338339	11	12
67	2016178008	11	14
68	2016111035	12	11
69	201236085	9	10
70	2014123008	14	14
71	201338064	13	14
72	201236263	13	12
73	2011126043	14	14
74	201237052	12	13
75	2014129030	13	13
76	2014112012	12	13
77	2014113004	10	13
78	201236806	10	12
79	2015105015	11	13
80	201236286	12	12
81	2016105052	12	10
82	201237645	10	14
83	2016120042	12	11
84	200934098	11	11
85	200934473	13	14
86	201035114	11	12
87	2015129031	13	13
88	2015101026	12	12
89	2015102052	14	13
90	201338185	13	15
91	2014127036	15	13
92	2016110054	14	12
93	2016102075	14	13
94	2016113038	13	11
95	201035639	14	12
96	2016114014	11	9
97	200527873	12	13
98	200832491	14	13
99	2016128009	13	13

ANEXO 3: Código fuente en C++

avl.h

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>

template <class T>
class avltree;

template <class T>
class avlnode
{
private:
        int fe;
        T dato;
        avlnode<T> *izq, *der;
public:
        avlnode();
        friend class avltree<T>;
};

template <class T>
avlnode<T>::avlnode()
{
        izq=NULL;
        der=NULL;
        fe=0;
}

template <class T>
class avltree
{
private:
        avlnode<T> *raiz;
        int comp;
public:
        avltree();
        avlnode<T>* regresaraiz();
        void inorden(avlnode<T> *);
        void preorden(avlnode<T> *);
        void posorden(avlnode<T> *);
        int altura(avlnode<T> *);
        int nivel(avlnode<T> *,T);
        int nodos(avlnode<T> *);
        void reinicia_comp();
        int retorna_comp();
        avlnode<T> * rotacionHD_HD(avlnode<T> *, avlnode<T> *);
        avlnode<T> * rotacionHI_HI(avlnode<T> *, avlnode<T> *);
        avlnode<T> * rotacionHD_HI(avlnode<T> *, avlnode<T> *);
        avlnode<T> * rotacionHI_HD(avlnode<T> *, avlnode<T> *);
        void reestructuraI(avlnode<T> *&, int *);
        void reestructuraD(avlnode<T> *&, int *);
        void inserta(avlnode<T> *, int *, T);
        int busca_avl(avlnode<T> *,T);


};

template <class T>
avltree<T>::avltree()
```

```
{
        raiz=NULL;
}

template <class T>
avlnode<T> * avltree<T>::regresaraiz()
{
        return raiz;
}

template <class T>
void avltree<T>::inorden(avlnode<T> *q)
{
        if(q!=NULL)
        {
                inorden(q->izq);
                cout<<q->dato<<"("<<q->fe<<")"<<" ";
                inorden(q->der);
        }
        else
                cout<<"* ";
}

template <class T>
void avltree<T>::preorden(avlnode<T> *q)
{
        if(q!=NULL)
        {
                cout<<q->dato<<"("<<q->fe<<")"<<" ";
                preorden(q->izq);
                preorden(q->der);

        }
        else
                cout<<"* ";
}

template <class T>
void avltree<T>::posorden(avlnode<T> *q)
{
        if(q!=NULL)
        {
                posorden(q->izq);
                posorden(q->der);
                cout<<q->dato<<"("<<q->fe<<")"<<" ";
        }
        else
                cout<<"* ";
}

template <class T>
int avltree<T>::nodos(avlnode<T> *q)
{
        if(q!=NULL)
                return 1+nodos(q->izq)+nodos(q->der);
        else
                return 0;
}

template <class T>
int avltree<T>::altura(avlnode<T> *q)
{
```

```
	if(q!=NULL)
	{
		if(altura(q->izq)>altura(q->der))
			return 1+altura(q->izq);
		else
			return 1+altura(q->der);
	}
	else
		return 0;
}

template <class T>
int avltree<T>::nivel(avlnode<T> *q,T clave)
{
	if(q!=NULL)
	{
		if(clave<q->dato)
			return 1+nivel(q->izq,clave);
		else
			if(clave>q->dato)
				return 1+nivel(q->der,clave);
			else
				return 1;
	}
	else
		return 0;
}

template <class T>
void avltree<T>::reinicia_comp()
{
	comp=0;
}

template <class T>
int avltree<T>::retorna_comp()
{
	return comp;
}

template <class T>
avlnode<T> * avltree<T>::rotacionHD_HD(avlnode<T> *ap, avlnode<T> *ap1)
{
	ap->der=ap1->izq;
	ap1->izq=ap;
	ap->fe=0;
	return(ap1);
}

template <class T>
avlnode<T> * avltree<T>::rotacionHI_HI(avlnode<T> *ap, avlnode<T> *ap1)
{
	ap->izq=ap1->der;
	ap1->der=ap;
	ap->fe=0;
	return(ap1);
}

template <class T>
avlnode<T> * avltree<T>::rotacionHD_HI(avlnode<T> *ap, avlnode<T> *ap1)
{
	avlnode<T> *ap2;
```

```
        ap2=ap1->izq;
        ap1->izq=ap2->der;
        ap2->der=ap1;
        ap->der=ap2->izq;
        ap2->izq=ap;
        if(ap2->fe==1)
                ap->fe=-1;
        else
                ap->fe=0;
        if(ap2->fe==-1)
                ap1->fe=1;
        else
                ap1->fe=0;
        return(ap2);
}

template <class T>
avlnode<T> * avltree<T>::rotacionHI_HD(avlnode<T> *ap, avlnode<T> *ap1)
{
        avlnode<T> *ap2;
        ap2=ap1->der;
        ap1->der=ap2->izq;
        ap2->izq=ap1;
        ap->izq=ap2->der;
        ap2->der=ap;
        if(ap2->fe==-1)
                ap->fe=1;
        else
                ap->fe=0;
        if(ap2->fe==1)
                ap1->fe=-1;
        else
                ap1->fe=0;
        return(ap2);
}

template <class T>
void avltree<T>::reestructuraI(avlnode<T> *&ap, int *cen)
{
        avlnode<T> *q, *r;
        if(*cen==1)
        {
                switch(ap->fe)
                {
                case -1:
                        ap->fe=0;
                        break;
                case 0:
                        ap->fe=1;
                        *cen=0;
                        break;
                case 1:
                        q=ap->der;
                        if(q->fe>=0)
                        {
                                ap->der=q->izq;
                                q->izq=ap;
                                switch(q->fe)
                                {
                                case 0:
                                        ap->fe=1;
                                        q->fe=-1;
```

```
                                        *cen=0;
                                        break;
                                case 1:
                                        ap->fe=0;
                                        q->fe=0;
                                        break;
                                }
                                ap=q;
                        }
                        else
                        {
                                ap=rotacionHD_HI(ap,q);
                                q->fe=0;
                        }
                        break;
                }
        }
}

template <class T>
void avltree<T>::reestructuraD(avlnode<T> *&ap, int *cen)
{
        avlnode<T> *q, *r;
        if(*cen==1)
        {
                switch(ap->fe)
                {
                case 1:
                        ap->fe=0;
                        break;
                case 0:
                        ap->fe=-1;
                        *cen=0;
                        break;
                case -1:
                        q=ap->izq;
                        if(q->fe<=0)
                        {
                                ap->izq=q->der;
                                q->der=ap;
                                switch(q->fe)
                                {
                                case 0:
                                        ap->fe=-1;
                                        q->fe=1;
                                        *cen=0;
                                        break;
                                case -1:
                                        ap->fe=0;
                                        q->fe=0;
                                        break;
                                }
                                ap=q;
                        }
                        else
                        {
                                ap=rotacionHI_HD(ap,q);
                                q->fe=0;
                        }
                        break;
                }
        }
```

```
}

template <class T>
void avltree<T>::inserta(avlnode<T> *ap, int *cen, T clave)
{
	avlnode<T> *q, *ap1, *ap2;
	if(ap!=NULL)
	{
		if(clave<ap->dato)
		{
			inserta(ap->izq,cen,clave);
			ap->izq=raiz;
			if(*cen==1)
			{
				switch(ap->fe)
				{
				case 1:
					ap->fe=0;
					*cen=0;
					break;
				case 0:
					ap->fe=-1;
					break;
				case -1:
					ap1=ap->izq;
					if(ap1->fe<=0)
					{
						ap1=rotacionHI_HI(ap,ap1);
						ap=ap1;
					}
					else
					{
						ap2=rotacionHI_HD(ap,ap1);
						ap=ap2;
					}
					ap->fe=0;
					*cen=0;
					break;
				}
			}
		}
		else
		{
			if(clave>ap->dato)
			{
				inserta(ap->der,cen,clave);
				ap->der=raiz;
				if(*cen==1)
				{
					switch(ap->fe)
					{
					case -1:
						ap->fe=0;
						*cen=0;
						break;
					case 0:
						ap->fe=1;
						break;
					case 1:
						ap1=ap->der;
						if(ap1->fe>=0)
						{
```

```
                                                                ap1=rotacionHD_HD(ap,ap1);
                                                                ap=ap1;
                                                        }
                                                        else
                                                        {
                                                                ap2=rotacionHD_HI(ap,ap1);
                                                                ap=ap2;
                                                        }
                                                        ap->fe=0;
                                                        *cen=0;
                                                        break;
                                                }
                                        }
                                }
                                else
                                {
                                        //Se intenta insertar un elemento repetido
                                }
                        }
                        raiz=ap;
                }
                else
                {
                        q=new(avlnode<T>);
                        q->dato=clave;
                        *cen=1;
                        raiz=q;
                }
}

template <class T>
int avltree<T>::busca_avl(avlnode<T> *q,T clave)
{
        if(q!=NULL)
        {
                comp++;
                if(clave<q->dato)
                        return busca_avl(q->izq,clave);
                else
                {
                        if(clave>q->dato)
                                return busca_avl(q->der,clave);
                        else
                                return 1;
                }
        }
        else
                return 0;
}
```

rb.h

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>

template <class T>
class rbtree;

template <class T>
class rbnode
```

```
{
private:
        T clave;
        char color;
        rbnode<T> *padre, *izq, *der;
public:
        rbnode();
        friend class rbtree<T>;
};

template <class T>
rbnode<T>::rbnode()
{
        padre=NULL;
        der=NULL;
        izq=NULL;
        color='r';
}

template <class T>
class rbtree
{
        rbnode<T> *raiz;
        rbnode<T> *q;
public :
        rbtree();
        rbnode<T>* regresaraiz();
        void inorden(rbnode<T> *);
   int nodos(rbnode<T> *);
        int altura(rbnode<T> *);
        int nivel(rbnode<T> *,T);
   bool verificar(int x);
        void insertarfix(rbnode<T> *,int);
        void insertar(T);
        int busca_rb(rbnode<T> *,T);
};

template <class T>
rbtree<T>::rbtree()
{
        q=NULL;
        raiz=NULL;
}

template <class T>
rbnode<T>* rbtree<T>::regresaraiz()
{
        return raiz;
}

template <class T>
void rbtree<T>::inorden(rbnode<T> *q)
{
        if(q!=NULL)
        {
                inorden(q->izq);
                cout<<q->clave<<"("<<q->color<<")"<<" ";
                inorden(q->der);
        }
}

template <class T>
```

```
int rbtree<T>::nodos(rbnode<T> *q)
{
	if(q!=NULL)
		return 1+nodos(q->izq)+nodos(q->der);
	else
		return 0;
}

template <class T>
int rbtree<T>::altura(rbnode<T> *q)
{
	if(q!=NULL)
	{
		if(altura(q->izq)>altura(q->der))
			return 1+altura(q->izq);
		else
			return 1+altura(q->der);
	}
	else
		return 0;
}

template <class T>
int rbtree<T>::nivel(rbnode<T> *q,T clave)
{
	if(q!=NULL)
	{
		if(clave<q->clave)
			return 1+nivel(q->izq,clave);
		else
			if(clave>q->clave)
				return 1+nivel(q->der,clave);
			else
				return 1;
	}
	else
		return 0;
}

template <class T>
void rbtree<T>::insertarfix(rbnode<T> *t,int h)
{
	int ladohijo;
	rbnode<T> *abuelo,*tio,*u;
	abuelo=t->padre;
	if(abuelo->izq!=NULL && abuelo->der!=NULL)
	{
		if(t==abuelo->izq)
			tio=abuelo->der;
		else
			tio=abuelo->izq;

		if(tio->color=='r')
		{
			tio->color='b';
			t->color='b';
			if(abuelo!=raiz)
				abuelo->color='r';
			if(abuelo->padre!=NULL)
			{
				u=abuelo->padre;
				if(u->izq==abuelo)ladohijo=1;
```

```
                                else ladohijo=2;
                                if(u->color=='r')insertarfix(u,ladohijo);
                        }
                        return;
                }
        }
        if(h==1 && abuelo->izq==t)
        {
                t->color='b'; abuelo->color='r';
                u=t->der;
                t->der=abuelo;
                t->padre=abuelo->padre;
                abuelo->padre=t;
                abuelo->izq=u;
                if(u!=NULL)
                        u->padre=abuelo;
                if(t->padre!=NULL)
                {
                        u=t->padre;
                        if(u->izq==t->der)
                                u->izq=t;
                        else
                                u->der=t;
                }
                else raiz=t;
        }
        else if(h==2 && abuelo->der==t)
        {
                t->color='b';
                abuelo->color='r';
                u=t->izq;
                t->izq=abuelo;
                t->padre=abuelo->padre;
                abuelo->padre=t;
                abuelo->der=u;
                if(u!=NULL)
                        u->padre=abuelo;
                if(t->padre!=NULL)
                {
                        u=t->padre;
                        if(u->izq==t->izq)
                                u->izq=t;
                        else
                                u->der=t;
                }
                else raiz=t;
        }
        else if(h==2 && abuelo->izq==t)
        {
                tio=t->der;
                u=tio->izq;
                abuelo->izq=tio;
                tio->padre=abuelo;
                tio->izq=t;
                t->padre=tio;
                t->der=u;
                if(u!=NULL)
                        u->padre=t;
                insertarfix(tio,1);
        }
        else if(h==1 && abuelo->der==t)
        {
```

```
                tio=t->izq;
                u=tio->der;
                abuelo->der=tio;
                tio->padre=abuelo;
                tio->der=t;
                t->padre=tio;
                t->izq=u;
                if(u!=NULL)
                        u->padre=t;
                insertarfix(tio,2);
        }
}

template <class T>
void rbtree<T>::insertar(T z)
{
        int ladohijo;
        rbnode<T> *s;
        rbnode<T> *q=new rbnode<T>;
        q->clave=z;
        if(raiz==NULL)
        {
                raiz=q;
                q->color='b';
        }
        else
        {
                s=raiz;
                int cen=0;
                while(cen==0)
                {
                        q->padre=s;
                        if(z<s->clave)
                        {
                                if(s->izq!=NULL)
                                        s=s->izq;
                                else
                                {
                                        s->izq=q; cen=1; ladohijo=1;
                                }
                        }
                        else
                        {
                                if(s->der)
                                        s=s->der;
                                else
                                {
                                        s->der=q; cen=1; ladohijo=2;
                                }
                        }
                }
                if(s->color=='r')
                        insertarfix(s,ladohijo);
        }
}

template <class T>
int rbtree<T>::busca_rb(rbnode<T> *q,T clave)
{
        if(q!=NULL)
        {
                if(clave<q->clave)
```

```
                        return busca_rb(q->izq,clave);
                else
                {
                        if(clave>q->clave)
                                return busca_rb(q->der,clave);
                        else
                                return 1;
                }
        }
        else
                return 0;
}
```

arreglo.h

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX 10000

template <class T>
class arreglo
{
private:
        int tam;
        T datos[MAX];
public:
        arreglo();
        void cargar();
        void inserta(T);
        void imprime();
        void busca(T);
        T regresa_valor(int);
        int regresa_tam();
        int aleatorio();
};

template <class T>
arreglo<T>::arreglo()
{
        tam=0;
}

template <class T>
void arreglo<T>::inserta(T valor)
{
        if(tam<MAX)
        {
                datos[tam]=valor;
                tam=tam+1;
        }
        else
        {
                cout<<"Arreglo lleno"<<endl;
        }
}

template <class T>
void arreglo<T>::imprime()
{
```

```
        if(tam>0)
        {
                for(int i=0;i<tam;i++)
                {
                        cout<<datos[i]<<endl;
                }
        }
}

template <class T>
void arreglo<T>::busca(T valor)
{
        int i;
        i=0;
        while((i<tam)&&(datos[i]!=valor))
        {
                i++;
        }
        if(i<tam)
        {
                cout<<setw(4)<<(i+1);
        }
        else
        {
                cout<<"NSE"<<endl;
        }
}

template <class T>
T arreglo<T>::regresa_valor(int ind)
{
        return datos[ind];
}

template <class T>
int arreglo<T>::regresa_tam()
{
        return tam;
}

template <class T>
int arreglo<T>::aleatorio()
{
        return(1 + rand()  % tam);
}
```

alumno.h

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

class alumno
{
private:
        char cod[15];
        int esc;
public:
        alumno();
        char * retorna_cod();
        void asignar(char[]);
```

```
        int operator > (alumno);
        int operator < (alumno);
        int operator != (alumno);
        friend istream &operator >> (istream &, alumno &);
        friend ostream &operator << (ostream &, alumno &);
};

alumno::alumno()
{
        esc=0;
}

char * alumno::retorna_cod()
{
        return cod;
}

void alumno::asignar(char cadena[])
{
        int i,j; char num[3];
        i=0; j=0;

        while (cadena[i]!=';'){
                cod[i]=cadena[i];
                i++;
        } cod[i]='\0'; i++;

        while (cadena[i]!='\0'){
                num[j]=cadena[i];
                i++; j++;
        } num[j]='\0'; esc=atoi(num);

}

int alumno::operator > (alumno objal)
{
        if(strcmp(cod,objal.cod)>0)
                return 1;
        else
                return 0;
}

int alumno::operator < (alumno objal)
{
        if(strcmp(cod,objal.cod)<0)
                return 1;
        else
                return 0;
}

int alumno::operator != (alumno objal)
{
        if(strcmp(cod,objal.cod)!=0)
        {
                return 1;
        }
        else
        {
                return 0;
        }
}
```

```
istream &operator >> (istream &lee, alumno &objal)
{
        cout<<"Ingrese codigo: "; lee>>objal.cod;
        cout<<"Ingrese especialidad: "; lee>>objal.esc;
        return lee;
}

ostream &operator << (ostream &escribe, alumno &objal)
{
        escribe<<"Codigo: "<<objal.cod<<endl;
        escribe<<"Especialidad: "<<objal.esc<<endl;
        return escribe;
}
```

aplicacion.cpp

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <fstream.h>
#include <iomanip.h>
#include <windows.h>
#include "alumno.h"
#include "arreglo.h"
#include "avl.h"
#include "rb.h"

double performancecounter_diff(LARGE_INTEGER *a, LARGE_INTEGER *b)
{
        LARGE_INTEGER freq;
        QueryPerformanceFrequency(&freq);
        return (double)(a->QuadPart - b->QuadPart) / (double)freq.QuadPart;
}

int main()
{
        arreglo<alumno> objarr;
        avltree<alumno> objavl;
        rbtree<alumno> objrb;
        int opc;

        alumno file;
        char buffer[50];
        ifstream archivo;
        ofstream salidacomp;
        ofstream salidatime;

        avlnode<alumno> *t;
        rbnode<alumno> *s;

        int op, m, i, j, n, cen, x;

        LARGE_INTEGER t_ini, t_fin;
        double secs,S;

        op=8;
        do
        {
                system("cls");
                cout<<"MENU DE OPCIONES " ;
                cout<<"\n 1. Insertar N registros";
```

```
                cout<<"\n 2. Mostrar arreglo";
                cout<<"\n 3. Mostrar arbol AVL";
                cout<<"\n 4. Mostrar arbol RB" ;
                cout<<"\n 5. Buscar un registro" ;
                cout<<"\n 6. Buscar N registros" ;
                cout<<"\n 0. Salir" ;
                cout<<endl;
                cout<<"\nElija una opcion: ";cin>>op;
                switch(op)
                {
                case 1:
                        i=0;
                        archivo.open("data2.csv");
                        if(archivo.good()!=true)
                        {
                                cout<<"El archivo no cargo correctamente";
                        }
                        srand(time(0));

                        //CARGA DE REGISTROS EN EL ARREGLO
                        QueryPerformanceCounter(&t_ini);
                        while(archivo.eof()!=true && i<MAX)
                        {
                                archivo.getline(buffer,50,'\n');
                                file.asignar(buffer);
                                objarr.inserta(file);
                                i++;
                        }
                        QueryPerformanceCounter(&t_fin);
                        secs = performancecounter_diff(&t_fin, &t_ini);
                        cout<<endl;

                        cout<<i<<" registros leidos del archivo"<<endl<<endl;
                        cout<<"Elementos en el arreglo: "<<objarr.regresa_tam()<<endl;
                        cout<<"Tiempo de insercion: "<<secs*1000000<<" microseg"<<endl;

                        //INSERCIÓN EN EL ARBOL AVL
                        QueryPerformanceCounter(&t_ini);
                        for(i=0;i<objarr.regresa_tam();i++)
                        {
                                cen=0;
                                t=objavl.regresaraiz();
                                objavl.inserta(t,&cen,objarr.regresa_valor(i));
                        }
                        QueryPerformanceCounter(&t_fin);
                        secs = performancecounter_diff(&t_fin, &t_ini);
                        cout<<endl;
                        t=objavl.regresaraiz();
                        cout<<"Total de nodos: "<<objavl.nodos(t)<<endl;
                        cout<<"Altura AVL: "<<objavl.altura(t)<<endl;
                        cout<<"Tiempo de insercion: "<<secs*1000000<<" microseg"<<endl;

                        //INSERCIÓN EN EL ÁRBOL RB
                        QueryPerformanceCounter(&t_ini);
                        for(i=0;i<objarr.regresa_tam();i++)
                        {
                                objrb.insertar(objarr.regresa_valor(i));
                        }
                        QueryPerformanceCounter(&t_fin);
                        secs = performancecounter_diff(&t_fin, &t_ini);
                        cout<<endl;
                        s=objrb.regresaraiz();
```

```
                        cout<<"Total de nodos: "<<objrb.nodos(s)<<endl;
                        cout<<"Altura RB: "<<objrb.altura(s)<<endl;
                        cout<<"Tiempo de insercion: "<<secs*1000000<<" microseg"<<endl;

                        cout<<endl;
                        system("pause");
                        break;
                case 2:
                        objarr.imprime();
                        cout<<endl;
                        system("pause");
                        break;
                case 3:
                        t=objavl.regresaraiz();
                        objavl.inorden(t);
                        cout<<endl;
                        system("pause");
                        break;
                case 4:
                        s=objrb.regresaraiz();
                        objrb.inorden(s);
                        cout<<endl;
                        system("pause");
                        break;
                case 5:
                        break;
                case 6:
                        cout<<"Ingrese la cantidad de busquedas: "; cin>>n;
                        i=1;
                        cout<<endl;
                        srand(time(0));
                        salidacomp.open("muestras_comp.csv");
                        salidatime.open("muestras_time.csv");

                        while(i<=n && i<=MAX)
                        {
                                m=objarr.aleatorio();

                                file=objarr.regresa_valor(m);

                                cout<<file.retorna_cod()<<"\t";
                                salidatime<<file.retorna_cod()<<";";
                                salidacomp<<file.retorna_cod()<<";";
                                cout<<"ARR: ";

                                QueryPerformanceCounter(&t_ini);
                                objarr.busca(file);
                                QueryPerformanceCounter(&t_fin);
                                secs = performancecounter_diff(&t_fin, &t_ini);
                                cout<<" - 
"<<setw(7)<<setiosflags(ios::fixed)<<setprecision(2)<<secs*1000000;
                                salidatime<<secs*1000000<<";";


                                objavl.reinicia_comp();
                                QueryPerformanceCounter(&t_ini);
                                t=objavl.regresaraiz();
                                objavl.busca_avl(t,file);
                                QueryPerformanceCounter(&t_fin);
                                secs = performancecounter_diff(&t_fin, &t_ini);
                                cout<<"\t\tAVL: "<<setw(2)<<objavl.retorna_comp()<<" - 
";
```

```
        cout<<setw(5)<<setiosflags(ios::fixed)<<setprecision(2)<<(secs)*1000000;
                                salidatime<<(secs)*1000000<<";";
                                salidacomp<<objavl.retorna_comp()<<";";

                                QueryPerformanceCounter(&t_ini);
                                s=objrb.regresaraiz();
                                x=objrb.nivel(s,file);
                                QueryPerformanceCounter(&t_fin);
                                secs = performancecounter_diff(&t_fin, &t_ini);
                                cout<<"\t\tRB: "<<setw(2)<<x<<" -
"<<setw(5)<<setiosflags(ios::fixed)<<setprecision(2)<<(secs)*1000000;
                                salidatime<<(secs)*1000000<<endl;
                                salidacomp<<x<<endl;

                                cout<<endl;
                                i++;
                        }

                        cout<<endl;
                        system("pause");
                        break;
                case 0:
                        break;
                default : cout<<"\nElija una opcion valida.";
                }
        } while(op!=0);

        cout<<endl;
        system("pause");
        return 0;
}
```