Manual Técnico

Juan Pablo de León Miranda

Estructura de Datos

6 de octubre de 2025

Notación BigO

Arbol AVL

Antes de poder empezar con nuestro análisis del árbol debemos de definir como es que se encuentra estructurado, conteniendo este las siguientes funciones y métodos a analizar:

- int altura(NodoAVL* nodo);
- int FactorEquilibrio(NodoAVL* nodo);
- NodoAVL* rotacionDerecha(NodoAVL* dato);
- NodoAVL* rotacionIzquierda(NodoAVL* dato);
- NodoAVL* insertar(NodoAVL* nodo, Libro libro);
- NodoAVL* valorMinimo(NodoAVL* nodo);
- NodoAVL* eliminar(NodoAVL* raiz, std::string titulo);
- void imprimirInOrder(NodoAVL* raiz);
- bool buscar(NodoAVL* raiz, std::string titulo);
- std::string conversionLowerCase(const std::string& string);
- Libro* busquedaLibroTitulo(NodoAVL* raiz, std::string titulo);
- AVL();
- void insertar(Libro libro);
- void eliminar(std::string titulo);
- bool buscar(std::string titulo);
- void imprimir();
- Libro* buscarLibro(std::string titulo);

1. Empezando por su constructor - AVL::AVL(): raiz(nullptr){}, que cuenta con una complejidad O(1), ya que solo inicializa un puntero
2. int altura(NodoAVL* nodo) – Donde su complejidad sigue siendo O(1), ya que nos da un acceso directo a un campo almacenado
3. int FactorEquilibrio(NodoAVL* nodo) – Sigue el mismo patron de tener una complejidad O(1) ya que hace uso de la función altura
4. NodoAVL* rotacionDerecha(NodoAVL* dato) y NodoAVL* rotacionIzquierda(NodoAVL* dato) – ambas son alanlizadas al mismo tiempo ya que su funcionamiento es parecido, únicamente que hacen las rotaciones en sentido contrario, además de que su complejidad es O(1), ya que únicamente hacen una reorganización de punteros.
5. NodoAVL* insertar(NodoAVL* nodo, Libro libro – Este si ya lleva a cambiar el tipo de notación ya que obtiene una complejidad O(log n), ya que para poder encontrar la posición donde insertar el nodo hace un recorrido con

complejidad O(log n), además de un máximo de rotación O(1) por inserción.

6. NodoAVL* valorMinimo(NodoAVL* nodo) – Tiene una complejidad O(log n) ya que hace un recorrido similar al de inserción, únicamente buscando el valor mas bajo que se encuentre mas a hacia la izquierda.
7. NodoAVL* eliminar(NodoAVL* raiz, std::string titulo) – Obtiene una complejidad O(log n), ya que para poder eliminar algún nodo, primero debe de hacer una búsqueda y esta tiene una complejidad O(log n), además de que al momento de hacer el reemplazo lo hace con un complejidad O(log n) para asi poder encontrar el valor mínimo en el subárbol derecho, junto con 1 rotacion máxima en caso de ser necesario con complejidad O(1).
8. bool buscar(NodoAVL* raiz, std::string titulo) – Tiene una complejidad O(log n).
9. Libro* busquedaLibroTitulo(NodoAVL* raiz, std::string titulo) – Al igual que buscar tiene una complejidad O(log n).
10. void imprimirInOrder(NodoAVL* raiz); - Tiene una complejidad O(n) ya que debe de recorrer rotos los nodos para poder mostrarlos.

Arbol B

De igual forma que con el árbol AVL, antes de poder hacer el análisis debemos de identificar las funciones y metetodos que lo estructuran, para asi poder llegar a la conclusión de su complejidad:

- BTreeNode::BTreeNode(int t1, bool leaf1)
- int BTreeNode::findKey(int fecha)
- void BTreeNode::imprimir()
- void BTree::insertion(Libro libro)
- void BTreeNode::insertNonFull(Libro libro)
- void BTreeNode::splitChild(int i, BTreeNode *y)
- void BTreeNode::deletion(int fecha)
- void BTreeNode::removeFromLeaf(int idx)
- void BTreeNode::removeFromNonLeaf(int idx)
- int BTreeNode::getPredecessor(int idx)
- int BTreeNode::getSuccessor(int idx)
- void BTreeNode::showRange(int low, int high)

1. BTreeNode::BTreeNode(int t1, bool leaf1) – Obtiene una complejidad O(t), donde t es proporcional la nivel del árbol que deseemos trabajar.
2. int BTreeNode::findKey(int fecha) – Tiene una complejidad O(t) ya que hace una búsqueda lineal dentro de las paginas (con máximo 2t-1)

3. void BTreeNode::imprimir() – Es de complejidad O(n) ya que debe de recorrer todo el árbol para poder imprimirlos.
4. void BTree::insertion(Libro libro) – Obtiene una complejidad O(t*log_t n), ya que para poder insertar el libro, ya que para sabre la altura se hace un recorrido de la raíz hasta la ultima hoja teniendo una complejidad O(log_t n), además de que se deben de hacer operaciones por nivel en cada nodo siendo una complejidad O(t), que haciendo la sumatoria nos da la complejidad inicial
5. void BTreeNode::insertNonFull(Libro libro) – Similar a la inserción anterior tiene una complejidad O(t*log_t n), ya que el funcionamiento es parecido.
6. void BTreeNode::splitChild(int i, BTreeNode *y) – Tiene una complejidad O(t) ya que realiza una copia de todos los t elementos encontrados en el array.
7. void BTreeNode::deletion(int fecha) , void BTreeNode::removeFromLeaf(int idx) y void BTreeNode::removeFromNonLeaf(int idx) – Los tres métodos de eliminación obtienen una complejidad O(t*log_t n), ya que primero hacen una búsqueda por niveles y como se explico previamente este tiene una complejidad O(log_t n), además de que realiza una reordenación al momento de eliminar el nodo siendo O(t) el máximo de las reordenaciones que debe realizar.
8. int BTreeNode::getPredecessor(int idx) y int BTreeNode::getSuccessor(int idx) – Ambos tienen una complejidad O(log_t n) ya que en el peor de los casos estos deben de recorrer dicha cantidad de veces el árbol.
9. void BTreeNode::showRange(int low, int high) – Tiene una complejidad O(k + log_t) ya que k varia dependiendo que tan amplia sea su rango de búsqueda.