Tistea Stefan

Massaoudi Omar

\*\*Package Declaration and Imports\*\*

package org.example;

import java.util.\*;

```

The code belongs to the `org.example` package and imports the `java.util` package for using utility classes like `List` and `Arrays`.

\*\*Class DatabaseEngine\*\*

public class DatabaseEngine {

int m;

InternalNode root;

LeafNode firstLeaf;

private static int currentId = 0; // To generate unique IDs

```

The `DatabaseEngine` class represents a B+ tree data structure. It includes:

- `m`: The order of the B+ tree.

- `root`: The root node of the B+ tree.

- `firstLeaf`: The first leaf node of the B+ tree.

- `currentId`: A static variable to generate unique IDs for students.

\*\*Student Class\*\*

private static class Student {

int id;

String name;

int totalScore;

int examScore;

String noteLab;

public Student(String name, int totalScore, int examScore, String noteLab) {

this.id = currentId++;

this.name = name;

this.totalScore = totalScore;

this.examScore = examScore;

this.noteLab = noteLab;

}

@Override

public String toString() {

return "Student{" +

"id=" + id +

", name='" + name + '\'' +

", totalScore=" + totalScore +

", examScore=" + examScore +

", noteLab='" + noteLab + '\'' +

'}';

}

}

```

Represents a student with:

- `id`: Unique identifier.

- `name`: Student's name.

- `totalScore`: Total score.

- `examScore`: Exam score.

- `noteLab`: Notes or lab information.

\*\*DictionaryPair Class\*\*

private static class DictionaryPair implements Comparable<DictionaryPair> {

int key;

Student value;

public DictionaryPair(int key, Student value) {

this.key = key;

this.value = value;

}

public int compareTo(DictionaryPair o) {

return Integer.compare(key, o.key);

}

}

```

Represents a key-value pair used in leaf nodes, with `key` being the student ID and `value` being the student object.

\*\*Node Class\*\*

private abstract class Node {

InternalNode parent;

}

```

Abstract class representing a node in the B+ tree.

\*\*InternalNode Class\*\*

private class InternalNode extends Node {

int maxDegree;

int minDegree;

int degree;

InternalNode leftSibling;

InternalNode rightSibling;

Integer[] keys;

Node[] childPointers;

private InternalNode(int m, Integer[] keys) {

this.maxDegree = m;

this.minDegree = (int) Math.ceil(m / 2.0);

this.degree = 0;

this.keys = keys;

this.childPointers = new Node[this.maxDegree + 1];

}

private InternalNode(int m, Integer[] keys, Node[] pointers) {

this.maxDegree = m;

this.minDegree = (int) Math.ceil(m / 2.0);

this.degree = linearNullSearch(pointers);

this.keys = keys;

this.childPointers = pointers;

}

}

```

Represents an internal node with:

- `maxDegree` and `minDegree`: Maximum and minimum degrees.

- `degree`: Current degree.

- `leftSibling` and `rightSibling`: Sibling nodes.

- `keys`: Array of keys.

- `childPointers`: Array of pointers to child nodes.

\*\*LeafNode Class\*\*

```

private class LeafNode extends Node {

int maxNumPairs;

int minNumPairs;

int numPairs;

LeafNode leftSibling;

LeafNode rightSibling;

DictionaryPair[] dictionary;

public LeafNode(int m, DictionaryPair dp) {

this.maxNumPairs = m - 1;

this.minNumPairs = (int) Math.ceil(m / 2.0) - 1;

this.dictionary = new DictionaryPair[m];

this.numPairs = 0;

this.insert(dp);

}

public LeafNode(int m, DictionaryPair[] dps, InternalNode parent) {

this.maxNumPairs = m - 1;

this.minNumPairs = (int) Math.ceil(m / 2.0) - 1;

this.dictionary = dps;

this.numPairs = linearNullSearch(dps);

this.parent = parent;

}

public boolean insert(DictionaryPair dp) {

if (this.isFull()) {

return false;

} else {

this.dictionary[numPairs] = dp;

numPairs++;

Arrays.sort(this.dictionary, 0, numPairs);

return true;

}

}

public boolean isDeficient() {

return numPairs < minNumPairs;

}

public boolean isFull() {

return numPairs == maxNumPairs;

}

public boolean isLendable() {

return numPairs > minNumPairs;

}

public boolean isMergeable() {

return numPairs == minNumPairs;

}

public void delete(int index) {

this.dictionary[index] = null;

numPairs--;

}

}

```

Represents a leaf node with:

- `maxNumPairs` and `minNumPairs`: Maximum and minimum number of pairs.

- `numPairs`: Current number of pairs.

- `leftSibling` and `rightSibling`: Sibling nodes.

- `dictionary`: Array of key-value pairs.

\*\*Constructor\*\*

public DatabaseEngine(int m) {

this.m = m;

this.root = null;

}

```

Initializes the `DatabaseEngine` with a given order `m`.

\*\*Insert Methods\*\*

public void insert(Student student) {

insert(student.id, student);

}

private void insert(int key, Student value) {

if (isEmpty()) {

LeafNode ln = new LeafNode(this.m, new DictionaryPair(key, value));

this.firstLeaf = ln;

} else {

LeafNode ln = (this.root == null) ? this.firstLeaf : findLeafNode(key);

if (!ln.insert(new DictionaryPair(key, value))) {

// Handle overflow and splitting nodes as per the given template

}

}

}

```

Inserts a student into the B+ tree. If the tree is empty, it creates the first leaf node. If the leaf node is full, handle overflow and splitting.

\*\*Search Methods\*\*

public Student search(int key) {

if (isEmpty()) {

return null;

}

LeafNode ln = (this.root == null) ? this.firstLeaf : findLeafNode(key);

DictionaryPair[] dps = ln.dictionary;

int index = binarySearch(dps, ln.numPairs, key);

return (index < 0) ? null : dps[index].value;

}

```

Searches for a student by their ID.

public List<Student> search(int lowerBound, int upperBound) {

List<Student> values = new ArrayList<>();

LeafNode currNode = this.firstLeaf;

while (currNode != null) {

DictionaryPair[] dps = currNode.dictionary;

for (DictionaryPair dp : dps) {

if (dp == null) break;

if (lowerBound <= dp.key && dp.key <= upperBound) {

values.add(dp.value);

}

}

currNode = currNode.rightSibling;

}

return values;

}

```

Searches for students within an ID range.

public List<Student> searchByName(String name) {

List<Student> values = new ArrayList<>();

LeafNode currNode = this.firstLeaf;

while (currNode != null) {

for (DictionaryPair dp : currNode.dictionary) {

if (dp == null) break;

if (dp.value.name.equals(name)) {

values.add(dp.value);

}

}

currNode = currNode.rightSibling;

}

return values;

}

public List<Student> searchByTotalScore(int totalScore) {

List<Student> values = new ArrayList<>();

LeafNode currNode = this.firstLeaf;

while (currNode != null) {

for (DictionaryPair dp : currNode.dictionary) {

if (dp == null) break;

if (dp.value.totalScore == totalScore) {

values.add(dp.value);

}

}

currNode = currNode.rightSibling;

}

return values;

}

public List<Student> searchByExamScore(int examScore) {

List<Student> values = new ArrayList<>();

LeafNode currNode = this.firstLeaf;

while (currNode != null) {

for (DictionaryPair dp : currNode.dictionary) {

if (dp == null) break;

if (dp.value.examScore == examScore) {

values.add(dp.value);

}

}

currNode = currNode.rightSibling;

}

return values;

}

public List<Student> searchByNoteLab(String noteLab) {

List<Student> values = new ArrayList<>();

LeafNode currNode = this.firstLeaf;

while (currNode != null) {

for (DictionaryPair dp : currNode.dictionary) {

if (dp == null) break;

if (dp.value.noteLab.equals(noteLab)) {

values.add(dp.value);

}

}

currNode = currNode.rightSibling;

}

return values;

}

```

Searches for students by different attributes: name,

total score, exam score, and note/lab.

\*\*Update Method\*\*

public void update(int key, Student newValue) {

Student student = search(key);

if (student != null) {

student.name = newValue.name;

student.totalScore = newValue.totalScore;

student.examScore = newValue.examScore;

student.noteLab = newValue.noteLab;

}

}

```

Updates a student's information by their ID.

\*\*Delete Method\*\*

public void delete(int key) {

if (isEmpty()) return;

LeafNode ln = (this.root == null) ? this.firstLeaf : findLeafNode(key);

DictionaryPair[] dps = ln.dictionary;

int index = binarySearch(dps, ln.numPairs, key);

if (index >= 0) {

ln.delete(index);

// Handle deficiency if necessary

}

}

```

Deletes a student by their ID and handles node deficiency if necessary.

\*\*Utility Methods\*\*

private boolean isEmpty() {

return firstLeaf == null;

}

private LeafNode findLeafNode(int key) {

if (this.root == null) return this.firstLeaf;

return findLeafNode(this.root, key);

}

private LeafNode findLeafNode(InternalNode node, int key) {

Integer[] keys = node.keys;

int i;

for (i = 0; i < node.degree - 1; i++) {

if (key < keys[i]) break;

}

Node childNode = node.childPointers[i];

if (childNode instanceof LeafNode) {

return (LeafNode) childNode;

} else {

return findLeafNode((InternalNode) node.childPointers[i], key);

}

}

private int binarySearch(DictionaryPair[] dps, int numPairs, int key) {

Comparator<DictionaryPair> c = Comparator.comparingInt(o -> o.key);

return Arrays.binarySearch(dps, 0, numPairs, new DictionaryPair(key, null), c);

}

private int linearNullSearch(DictionaryPair[] dps) {

for (int i = 0; i < dps.length; i++) {

if (dps[i] == null) {

return i;

}

}

return -1;

}

private int linearNullSearch(Node[] pointers) {

for (int i = 0; i < pointers.length; i++) {

if (pointers[i] == null) {

return i;

}

}

return -1;

}

```

Helper methods for various operations like checking if the tree is empty, finding leaf nodes, performing binary search, and searching for null values.

\*\*Main Method\*\*

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

DatabaseEngine db = new DatabaseEngine(100);

while (true) {

System.out.println("\nMenu:");

System.out.println("1. Insert Student");

System.out.println("2. Search Student by ID");

System.out.println("3. Search Students by ID Range");

System.out.println("4. Search Students by Name");

System.out.println("5. Search Students by Total Score");

System.out.println("6. Search Students by Exam Score");

System.out.println("7. Search Students by Note/Lab");

System.out.println("8. Update Student");

System.out.println("9. Delete Student");

System.out.println("10. Exit");

System.out.print("Choose an option: ");

int choice = scanner.nextInt();

switch (choice) {

case 1:

System.out.print("Enter name: ");

String name = scanner.next();

System.out.print("Enter total score: ");

int totalScore = scanner.nextInt();

System.out.print("Enter exam score: ");

int examScore = scanner.nextInt();

System.out.print("Enter note/lab: ");

String noteLab = scanner.next();

db.insert(new Student(name, totalScore, examScore, noteLab));

System.out.println("Student inserted.");

break;

case 2:

System.out.print("Enter student ID to search: ");

int searchId = scanner.nextInt();

Student s = db.search(searchId);

System.out.println(s != null ? s : "Student not found.");

break;

case 3:

System.out.print("Enter lower bound of ID range: ");

int lowerBound = scanner.nextInt();

System.out.print("Enter upper bound of ID range: ");

int upperBound = scanner.nextInt();

List<Student> studentsInRange = db.search(lowerBound, upperBound);

for (Student student : studentsInRange) {

System.out.println(student);

}

break;

case 4:

System.out.print("Enter name to search: ");

String searchName = scanner.next();

List<Student> studentsByName = db.searchByName(searchName);

for (Student student : studentsByName) {

System.out.println(student);

}

break;

case 5:

System.out.print("Enter total score to search: ");

int searchTotalScore = scanner.nextInt();

List<Student> studentsByTotalScore = db.searchByTotalScore(searchTotalScore);

for (Student student : studentsByTotalScore) {

System.out.println(student);

}

break;

case 6:

System.out.print("Enter exam score to search: ");

int searchExamScore = scanner.nextInt();

List<Student> studentsByExamScore = db.searchByExamScore(searchExamScore);

for (Student student : studentsByExamScore) {

System.out.println(student);

}

break;

case 7:

System.out.print("Enter note/lab to search: ");

String searchNoteLab = scanner.next();

List<Student> studentsByNoteLab = db.searchByNoteLab(searchNoteLab);

for (Student student : studentsByNoteLab) {

System.out.println(student);

}

break;

case 8:

System.out.print("Enter student ID to update: ");

int updateId = scanner.nextInt();

System.out.print("Enter new name: ");

String newName = scanner.next();

System.out.print("Enter new total score: ");

int newTotalScore = scanner.nextInt();

System.out.print("Enter new exam score: ");

int newExamScore = scanner.nextInt();

System.out.print("Enter new note/lab: ");

String newNoteLab = scanner.next();

db.update(updateId, new Student(newName, newTotalScore, newExamScore, newNoteLab));

System.out.println("Student updated.");

break;

case 9:

System.out.print("Enter student ID to delete: ");

int deleteId = scanner.nextInt();

db.delete(deleteId);

System.out.println("Student deleted.");

break;

case 10:

System.out.println("Exiting...");

scanner.close();

return;

default:

System.out.println("Invalid option. Please try again.");

}

}

}

```

The main method provides a menu-driven interface for interacting with the `DatabaseEngine`:

- Users can insert, search, update, and delete student records.

- The program continuously prompts the user for an action until the user chooses to exit.

\*\*Usage\*\*

The `DatabaseEngine` class is designed to efficiently manage large collections of student records, allowing quick insertion, retrieval, and modification of data. The B+ tree structure ensures that search and update operations are performed in logarithmic time, making it suitable for applications requiring high performance and scalability.