ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΑΤΡΩΝ
UNIVERSITY OF PATRAS

PROJECT ΔΟΜΩΝ ΔΕΔΟΜΕΝΩΝ 2023_4

Μέλη

- ΒΕΡΥΚΙΟΣ ΑΓΓΕΛΟΣ 1100500
- ΒΟΓΙΑΝΤΖΗΣ ΑΝΑΣΤΑΣΙΟΣ 1100506
- ΚΟΛΥΒΡΑΣ ΚΩΝΣΤΑΝΤΙΝΟΣ 1103826
- ΠΑΠΑΚΩΝΣΤΑΝΤΙΝΟΥ ΣΤΑΜΑΤΗΣ 1100669

Περιεχόμενα

Εισαγωγή

Κώδικας ανάγνωσης αρχείου

```
    public static List<entry> readFromFile(String filePath) {
            List<entry> dataList = new ArrayList<>();
            try {
                List<String> lines =
Files.readAllLines(Paths.get("data.txt"));
                for (String line : lines.subList(1, lines.size())) {
                    String[] parts = line.split(",", 4);
                    int period = Integer.parseInt(parts[0]);
                    String Birth_Death = parts[1];
                    String region = parts[2].replace("\"", "");
                    int count = Integer.parseInt(parts[3]);
                    dataList.add(new entry(period, Birth_Death, region,
count));
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
            return dataList;
    }
```

Παρατηρήσεις:

Αυτό το κομμάτι κώδικα διαβάζει ένα αρχείο με όνομα "data.txt" και αποθηκεύει τις γραμμές του σε μια λίστα. Επεξεργάζεται κάθε γραμμή, παραλείποντας την πρώτη, και χωρίζει τα δεδομένα σε τέσσερα μέρη: period, Birth_Death, region και count. Δημιουργεί αντικείμενα τύπου entry με αυτά τα δεδομένα και τα προσθέτει σε μια λίστα, την οποία επιστρέφει στο τέλος.

Κώδικας εισαγωγής στοιχείων σε hash table

```
    public static Map<String, Integer> sumBirthsByRegion(List<entry>
dataList) {

            Map<String, Integer> birthsByRegion = new HashMap<>();
```

```
            for (entry entry : dataList) {
                if (entry.getBirth_Death().equals("Births")) {
                    if (birthsByRegion.containsKey(entry.getRegion()))
{
                        birthsByRegion.put(entry.getRegion(),
birthsByRegion.get(entry.getRegion()) + entry.getCount());
                    } else {
                        birthsByRegion.put(entry.getRegion(),
entry.getCount());
                    }
                }
            }
        return birthsByRegion;
    }
```

Παρατηρήσεις:

Αυτό το κομμάτι κώδικα δημιουργεί ένα hash table για να αποθηκεύσει τον συνολικό αριθμό γεννήσεων ανά περιοχή. Για κάθε αντικείμενο στη λίστα δεδομένων, ελέγχει αν το πεδίο "Birth_Death" έχει την τιμή "Births" και προσθέτει τον αριθμό γεννήσεων στο hash table. Επιστρέφει το hash table που περιέχει τον συνολικό αριθμό γεννήσεων για κάθε περιοχή.

Κώδικας Scanner και εκτύπωσης hash tables

```
        Scanner scanner = new Scanner(System.in);
        System.out.print("Type 0 for MergeSort or 1 for QuickSort: ");
        String sort = scanner.nextLine();
        scanner.close();

        if (sort.equals("0")) {
            List<Map.Entry<String, Integer>> sortedList =
mergeSortRegions(entryList);
            for (Map.Entry<String, Integer> entry : sortedList) {
                System.out.println("Region: " + entry.getKey() + ",
Total Births: " + entry.getValue());
            }
        } else if (sort.equals("1")) {
            List<Map.Entry<String, Integer>> sortedList =
getQuickSortedRegions(entryList);
            for (Map.Entry<String, Integer> entry : sortedList) {
                System.out.println("Region: " + entry.getKey() + ",
Total Births: " + entry.getValue());
```

```
            }
        } else {
            System.out.println("Error");
        }
```

Παρατηρήσεις:

Αυτό το κομμάτι κώδικα ζητά από τον χρήστη να επιλέξει μεταξύ MergeSort και Quick Sort μέσω της εισόδου από το πληκτρολόγιο. Ανάλογα με την επιλογή, καλεί την αντίστοιχη μέθοδο ταξινόμησης για μια λίστα περιοχών και γεννήσεων. Στη συνέχεια, εμφανίζει τα αποτελέσματα ταξινόμησης ή εμφανίζει μήνυμα σφάλματος αν η είσοδος δεν είναι έγκυρη.

PART I: “Sorting and Searching Algorithms”

Merge Sort και Quick Sort

Merge Sort Algorithm (Regions→ Birth_Count)

```
    public static List<Map.Entry<String, Integer>>
mergeSortRegions(List<Map.Entry<String, Integer>> entryList) {
        if (entryList.size() <= 1) {
            return entryList;
        }

        int mid = entryList.size() / 2;

        List<Map.Entry<String, Integer>> left = new
ArrayList<>(entryList.subList(0, mid));
        List<Map.Entry<String, Integer>> right = new
ArrayList<>(entryList.subList(mid, entryList.size()));

        return merge(mergeSortRegions(left), mergeSortRegions(right));
    }

    private static List<Map.Entry<String, Integer>>
merge(List<Map.Entry<String, Integer>> left, List<Map.Entry<String,
Integer>> right) {
        List<Map.Entry<String, Integer>> merged = new ArrayList<>();
        int leftIndex = 0, rightIndex = 0;

        while (leftIndex < left.size() && rightIndex < right.size()) {
            if (left.get(leftIndex).getValue() <=
right.get(rightIndex).getValue()) {
                merged.add(left.get(leftIndex++));
            } else {
                merged.add(right.get(rightIndex++));
            }
        }

        while (leftIndex < left.size()) {
            merged.add(left.get(leftIndex++));
        }

        while (rightIndex < right.size()) {
            merged.add(right.get(rightIndex++));
        }
```

```
        return merged;
    }
```

Παρατηρήσεις:

Αυτό το κομμάτι κώδικα εφαρμόζει τον αλγόριθμο Merge Sort για ταξινόμηση της λίστας με εισόδους περιοχών και αριθμών γεννήσεων. Η μέθοδος mergeSortRegions διαιρεί τη λίστα εισόδου σε δύο υπολίστες, αναδρομικά εφαρμόζει τον εαυτό της σε κάθε υπολίστα, και τελικά συγχωνεύει τις ταξινομημένες υπολίστες με τη μέθοδο merge. Ο αλγόριθμος merge συγκρίνει στοιχεία από δύο υπολίστες και τα τοποθετεί σωστά σε μια νέα λίστα.

Quick Sort (Regions→ Count)

```
    public static void quickSortRegions(List<Map.Entry<String,
Integer>> entryList, int low, int high) {
        if (low < high) {
            int pi = partition(entryList, low, high);

            quickSortRegions(entryList, low, pi - 1);
            quickSortRegions(entryList, pi + 1, high);
        }
    }

    private static int partition(List<Map.Entry<String, Integer>>
entryList, int low, int high) {
        int pivot = entryList.get(high).getValue();
        int i = (low - 1);

        for (int j = low; j < high; j++) {
            if (entryList.get(j).getValue() <= pivot) {
                i++;

                Map.Entry<String, Integer> temp = entryList.get(i);
                entryList.set(i, entryList.get(j));
                entryList.set(j, temp);
            }
        }

        Map.Entry<String, Integer> temp = entryList.get(i + 1);
        entryList.set(i + 1, entryList.get(high));
        entryList.set(high, temp);

        return i + 1;
    }
```

Παρατηρήσεις:

Αυτό το κομμάτι κώδικα εφαρμόζει τον αλγόριθμο Quick Sort για ταξινόμηση της λίστας με εισόδους περιοχών και αριθμών γεννήσεων. Η μέθοδος quickSortRegions διαιρεί τη λίστα εισόδου με βάση τον δείκτη προς το χαμηλότερο και τον υψηλότερο δείκτη, εφαρμόζει αναδρομικά τον εαυτό της για κάθε υπολίστα που προκύπτει μετά την κάθε διαίρεση, και τελικά τα ταξινομημένα τμήματα συνδυάζονται μέσω της μεθόδου partition. Η partition διαχωρίζει τη λίστα σε δύο υπολίστες με βάση ένα στοιχείο προς έναν πίνακα με μικρότερα και ένα μεγαλύτερο από το στοιχείο αυτό.

- Συγκρίνατε πειραματικά τους δύο (2) αλγορίθμους. Τι παρατηρείτε;

Ο mergeSort είναι γρηγορότερος και αυτό επιβεβαιώνεται και από την πολυπλοκότητα του η οποία είναι O(nlogn) σε αντίθεση με τον quick sort ο οποίος έχει την ίδια πολυπλοκότητα μόνο στην καλύτερη περίπτωση ενώ μπορεί να φτάσει έως και O(n^2) .

Heap Sort και Counting Sort

Heap Sort Algorithm (Regions → Death_count)

```
    public static List<Map.Entry<String, Integer>>
heapSortRegions(List<Map.Entry<String, Integer>> entryList) {
        int n = entryList.size();

        for (int i = n / 2 - 1; i >= 0; i--) {
            heapify(entryList, n, i);
        }

        for (int i = n - 1; i >= 0; i--) {
            Map.Entry<String, Integer> temp = entryList.get(0);
            entryList.set(0, entryList.get(i));
            entryList.set(i, temp);

            heapify(entryList, i, 0);
        }

        return entryList;
    }
```

```
    private static void heapify(List<Map.Entry<String, Integer>>
entryList, int n, int i) {
        int largest = i;
        int left = 2 * i + 1;
        int right = 2 * i + 2;

        if (left < n && entryList.get(left).getValue() >
entryList.get(largest).getValue()) {
            largest = left;
        }

        if (right < n && entryList.get(right).getValue() >
entryList.get(largest).getValue()) {
            largest = right;
        }

        if (largest != i) {
            Map.Entry<String, Integer> swap = entryList.get(i);
            entryList.set(i, entryList.get(largest));
            entryList.set(largest, swap);

            heapify(entryList, n, largest);
        }
    }
```

Παρατηρήσεις:

Αυτό το κομμάτι κώδικα υλοποιεί τον αλγόριθμο Heap Sort για ταξινόμηση μιας λίστας περιοχών και αριθμών γεννήσεων. Ο αλγόριθμος αυτός δημιουργεί ένα max heap από τα δεδομένα και στη συνέχεια επαναλαμβάνει τον ακόλουθο βήμα: ανταλλάσσει το πρώτο στοιχείο (το μεγαλύτερο) με το τελευταίο στοιχείο, μειώνει το μέγεθος του heap κατά ένα, και στη συνέχεια καλεί τη διαδικασία heapify για να επανορθώσει την ιδιότητα του max heap. Οι διαδικασίες heapify και heapSortRegions υλοποιούν αντίστοιχα τη λειτουργία αυτών των βημάτων.

Counting Sort Algorithm (Regions → Death_count)

```
  public static List<Map.Entry<String, Integer>>
countingSortRegions(List<Map.Entry<String, Integer>> entryList) {
        int max = findMax(entryList);
        int[] count = new int[max + 1];
        List<Map.Entry<String, Integer>> output = new
ArrayList<>(entryList.size());
        for (int i = 0; i < entryList.size(); i++) {
            output.add(null);
        }
        for (int i = 0; i <= max; i++) {
```

```
            count[i] = 0;
        }
        for (Map.Entry<String, Integer> entry : entryList) {
            count[entry.getValue()]++;
        }

        for (int i = 1; i <= max; i++) {
            count[i] += count[i - 1];
        }

        for (int i = entryList.size() - 1; i >= 0; i--) {
            Map.Entry<String, Integer> entry = entryList.get(i);
            output.set(count[entry.getValue()] - 1, entry);
            count[entry.getValue()]--;
        }

        return output;
    }

    private static int findMax(List<Map.Entry<String, Integer>>
entryList) {
        int max = entryList.get(0).getValue();
        for (Map.Entry<String, Integer> entry : entryList) {
            if (entry.getValue() > max) {
                max = entry.getValue();
            }
        }
        return max;
    }
```

Παρατηρήσεις:

Αυτό το κομμάτι κώδικα εφαρμόζει τον αλγόριθμο Counting Sort για ταξινόμηση μιας λίστας περιοχών και αριθμών γεννήσεων. Ξεκινά εντοπίζοντας τον μέγιστο αριθμό γεννήσεων στη λίστα εισόδου. Στη συνέχεια, δημιουργεί έναν πίνακα μετρητών για κάθε διαφορετική τιμή αριθμού γεννήσεων. Καταμετράει πόσες φορές εμφανίζεται κάθε τιμή αυτού του αριθμού γεννήσεων στη λίστα. Στη συνέχεια, χρησιμοποιεί αυτές τις πληροφορίες για να τοποθετήσει κάθε στοιχείο στην κατάλληλη θέση στην τελική ταξινομημένη λίστα. Τέλος, επιστρέφει την ταξινομημένη λίστα.

- Συγκρίνατε πειραματικά τους δύο (2) αλγορίθμους. Τι παρατηρείτε;

Ο αλγόριθμος Counting Sort είναι περίπου 2-3 φορές πιο γρήγορος από τον Heap Sort.Σίγουρα το γεγονός του ότι η λίστα περιέχει ένα hash table στο οποίο εκεί είναι αποθηκευμένο το count_birth επιβραδύνει την υπόθεση.

Δυαδική Αναζήτηση και Αναζήτηση με Παρεμβολή

Δυαδική Αναζήτηση (Regions→ Birth_Count)

```
    public static List<Map.Entry<String, Integer>>
binarySearchRegions(List<Map.Entry<String, Integer>> entryList, int b1,
int b2) {
        List<Map.Entry<String, Integer>> result = new ArrayList<>();
        int left = 0, right = entryList.size() - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2;
            int midValue = entryList.get(mid).getValue();

            if (midValue >= b1 && midValue <= b2) {
                result.add(entryList.get(mid));
                int temp = mid - 1;
                while (temp >= 0 && entryList.get(temp).getValue() >=
b1 && entryList.get(temp).getValue() <= b2) {
                    result.add(entryList.get(temp--));
                }
                temp = mid + 1;
                while (temp < entryList.size() &&
entryList.get(temp).getValue() >= b1 && entryList.get(temp).getValue()
<= b2) {
                    result.add(entryList.get(temp++));
                }
                break;
            } else if (midValue < b1) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }

        return result;
    }
```

Παρατηρήσεις:

Αυτό το κομμάτι κώδικα ξεκινάει με την αρχικοποίηση δύο δεικτών, έναν για την αρχή της λίστας και έναν για το τέλος. Στη συνέχεια, επαναλαμβάνει την αναζήτηση όσο ο δείκτης αρχής είναι μικρότερος ή ίσος με τον δείκτη τέλους. Σε κάθε βήμα, υπολογίζει τον δείκτη μέσης τιμής και ελέγχει το στοιχείο στη μέση της λίστας. Εάν η τιμή του στοιχείου είναι εντός του καθορισμένου εύρους, το προσθέτει στη λίστα αποτελεσμάτων. Έπειτα, επεκτείνει την αναζήτηση στις δύο υπολίστες αριστερά και δεξιά από το στοιχείο μέσης τιμής. Αυτή η διαδικασία επαναλαμβάνεται μέχρι να εντοπιστούν όλα τα στοιχεία που πληρούν την συνθήκη του εύρους. Τέλος, επιστρέφει την λίστα με τα εντοπισμένα στοιχεία.

Αναζήτηση με παρεμβολή (Regions→ Birth_Count)

```
    public static List<Map.Entry<String, Integer>>
interpolationSearchRegions(List<Map.Entry<String, Integer>> entryList,
int b1, int b2) {
        List<Map.Entry<String, Integer>> result = new ArrayList<>();
        int low = 0;
        int high = entryList.size() - 1;

        while (low <= high && b1 <= b2 && b1 <=
entryList.get(high).getValue() && b2 >= entryList.get(low).getValue())
{
            if (entryList.get(high).getValue() ==
entryList.get(low).getValue()) {
                if (entryList.get(low).getValue() >= b1 &&
entryList.get(low).getValue() <= b2) {
                    for (int i = low; i <= high; i++) {
                        result.add(entryList.get(i));
                    }
                }
                break;
            }

            int pos = low + (int) (((double) (high - low) /
(entryList.get(high).getValue() - entryList.get(low).getValue())) * (b1
- entryList.get(low).getValue()));

            if (pos < low || pos > high) {
                break;
            }

            int posValue = entryList.get(pos).getValue();
```

```
            if (posValue >= b1 && posValue <= b2) {
                result.add(entryList.get(pos));

                int temp = pos - 1;
                while (temp >= low && entryList.get(temp).getValue() >=
b1 && entryList.get(temp).getValue() <= b2) {
                    result.add(entryList.get(temp));
                    temp--;
                }

                temp = pos + 1;
                while (temp <= high && entryList.get(temp).getValue()
>= b1 && entryList.get(temp).getValue() <= b2) {
                    result.add(entryList.get(temp));
                    temp++;
                }

                break;
            } else if (posValue < b1) {
                low = pos + 1;
            } else {
                high = pos - 1;
            }
        }
```

Παρατηρήσεις:

Σε αυτό το κομμάτι κώδικα ορίζονται δύο δείκτες low και high που αντιστοιχούν στην αρχή και το τέλος της λίστας αντίστοιχα. Στη συνέχεια, επαναλαμβάνεται η αναζήτηση όσο το low είναι μικρότερο ή ίσο του high, ενώ το εύρος b1 έως b2 είναι έγκυρο και εμπίπτει εντός των τιμών της λίστας. Κάθε επανάληψη υπολογίζει τη θέση pos στην οποία υποτίθεται ότι βρίσκεται το στοιχείο που αναζητούμε, χρησιμοποιώντας τον τύπο παρεμβολής. Στη συνέχεια, ελέγχει αν η τιμή του στοιχείου στη θέση pos εμπίπτει εντός του εύρους b1 έως b2, και αν ναι, το προσθέτει στη λίστα αποτελεσμάτων. Επίσης, επεκτείνει την αναζήτηση στις υπολίστες αριστερά και δεξιά από τη θέση pos. Η διαδικασία αυτή συνεχίζεται μέχρι να εντοπιστούν όλα τα στοιχεία που πληρούν την συνθήκη του εύρους. Τέλος, επιστρέφεται η λίστα με τα εντοπισμένα στοιχεία.

- Τί παρατηρείτε ως προς τους χρόνους μέσης περίπτωσης; Πόσο η κατανομή του Data Set επηρεάζει την απόδοση του κάθε αλγορίθμου;

  Όσον αφορά τον χρόνο της μέσης περίπτωσης ο αλγόριθμος Interpolation search είναι γρηγορότερος από τον Binary Search.

Δυικής Αναζήτησης Παρεμβολής (BIS)

Δυικής Αναζήτησης Παρεμβολής (BIS) (Regions→ Birth_Count)

```
    public static List<Map.Entry<String, Integer>>
binaryInterpolationSearchRegions(List<Map.Entry<String, Integer>>
entryList, int b1, int b2) {
        List<Map.Entry<String, Integer>> result = new ArrayList<>();
        int left = 0;
        int right = entryList.size() - 1;
        int size = right - left + 1;
        int next;

        System.out.println("Starting binary interpolation search...");
        System.out.println("Initial values: left = " + left + ", right
= " + right + ", b1 = " + b1 + ", b2 = " + b2);
        System.out.println("entryList.get(left).getValue() = " +
entryList.get(left).getValue() + ", entryList.get(right).getValue() = "
+ entryList.get(right).getValue());

        while (size > 3) {
            int denominator = entryList.get(right).getValue() -
entryList.get(left).getValue();
            if (denominator != 0) {
                next = left + ((right - left) * (b1 -
entryList.get(left).getValue())) / denominator;
            } else {
                next = left;
            }

            if (next < 0) next = 0;
            if (next >= entryList.size()) next = entryList.size() - 1;

            System.out.println("Calculated next = " + next + ",
entryList.get(next).getValue() = " + entryList.get(next).getValue());

            if (b1 >= entryList.get(next).getValue()) {
                int i = 1;
                while (next + i * (int) Math.sqrt(size - 1) <
entryList.size() && b1 > entryList.get(next + i * (int) Math.sqrt(size
- 1)).getValue()) {
                    i++;
                }
                right = Math.min(next + i * (int) Math.sqrt(size),
entryList.size() - 1);
                left = next + (i - 1) * (int) Math.sqrt(size);
```

```
            } else {
                int i = 1;
                while (next - i * (int) Math.sqrt(size - 1) >= 0 && b1
< entryList.get(next - i * (int) Math.sqrt(size - 1)).getValue()) {
                    i++;
                }
                right = next - (i - 1) * (int) Math.sqrt(size);
                left = Math.max(next - i * (int) Math.sqrt(size), 0);
            }

            System.out.println("Updated values: left = " + left + ",
right = " + right);
            size = right - left + 1;
            System.out.println("Updated size = " + size);
        }

        for (int j = left; j <= right; j++) {
            if (entryList.get(j).getValue() >= b1 &&
entryList.get(j).getValue() <= b2) {
                result.add(entryList.get(j));
            }
        }

        return result;
    }
```

Παρατηρήσεις:

Αυτό το τμήμα κώδικα υλοποιεί τον αλγόριθμο BIS, ο οποίος συνδυάζει τη δυαδική αναζήτηση με την παρεμβολή για την αναζήτηση σε μια ταξινομημένη λίστα περιοχών και αριθμών γεννήσεων που εμπίπτουν μέσα σε ένα καθορισμένο εύρος. Αρχικά, υπολογίζει μια προσεγγιστική θέση next χρησιμοποιώντας την τεχνική της παρεμβολής. Στη συνέχεια, επεκτείνει το εύρος αναζήτησης με βάση το εύρος που υπολογίστηκε και ενημερώνει τα όρια left και right. Αυτή η διαδικασία επαναλαμβάνεται μέχρι οι δείκτες να βρίσκονται σε απόσταση τριών στοιχείων ή λιγότερο. Τέλος, ελέγχει τα στοιχεία στο εύρος που καθορίστηκε και τα προσθέτει στη λίστα αποτελεσμάτων εάν πληρούν την συνθήκη του εύρους. Αυτός ο αλγόριθμος συνδυάζει την ταχύτητα της δυαδικής αναζήτησης με την ακρίβεια της παρεμβολής για αποτελεσματική αναζήτηση εντός ενός καθορισμένου εύρους.

PART II: “BSTs && HASHING”

Secondary classes

Entry class

```
class Entry {
    private int period;
    private String birthDeath;
    private String region;
    private int count;

    public Entry(int period, String birthDeath, String region, int
count) {
        this.period = period;
        this.birthDeath = birthDeath;
        this.region = region;
        this.count = count;
    }

    public int getPeriod() {
        return period;
    }

    public String getBirthDeath() {
        return birthDeath;
    }

    public String getRegion() {
        return region;
    }

    public int getCount() {
        return count;
    }

    public void setCount(int count) {
        this.count = count;
    }

    public String toString() {
```

```
        return "Period: " + period + ", Birth/Death: " + birthDeath +
", Region: " + region + ", Count: " + count;
    }
}
```

AVL Node Class

```
import java.util.ArrayList;
import java.util.List;

class AVLNode {
    String key;
    List<Entry> entries;
    int height;
    AVLNode left, right;

    public AVLNode(String key, Entry entry) {
        this.key = key;
        this.entries = new ArrayList<>();
        this.entries.add(entry);
        this.height = 1;
    }
}
```

AVL με Regions

Menu

```
        Scanner scanner = new Scanner(System.in);

        while (true) {
            System.out.println("Menu:");
            System.out.println("1. Print AVL tree");
            System.out.println("2. Search for count of births");
            System.out.println("3. Modify the amount of births");
            System.out.println("4. Delete data for a region and
period");
            System.out.println("5. Exit");
            System.out.print("Enter your choice: ");
            int choice = scanner.nextInt();
            scanner.nextLine();
```

Pre-Order Func

```
    public void preOrder(AVLNode node, int level) {
        if (node != null) {
            printIndent(level);
```

```
            System.out.println(node.key + " (" + node.entries.size() +
" entries)");
            for (Entry entry : node.entries) {
                printIndent(level + 1);
                System.out.println(entry);
            }
            preOrder(node.left, level + 1);
            preOrder(node.right, level + 1);
        }
    }
```

Παρατηρήσεις:

Αυτή η μέθοδος preOrder εκτελεί μια προδιατεταγμένη διάσχιση σε ένα δέντρο AVL, ξεκινώντας από την ρίζα. Κατά τη διάσχιση, εκτυπώνει τα κλειδιά των κόμβων και τον αριθμό των καταχωρήσεων που περιέχονται σε κάθε κόμβο, καθώς και τις καταχωρήσεις ίδιων περιοχών που υπάρχουν σε κάθε κόμβο. Στη συνέχεια, καλεί τον εαυτό της για το αριστερό παιδί του κόμβου και στη συνέχεια για το δεξιό παιδί του κάθε κόμβου, αυξάνοντας το επίπεδο κατά ένα για κάθε επόμενο επίπεδο στο δέντρο.

Insert Method (Με συναρτήσεις για να είναι ισοζυγισμένο)

```
    public AVLNode insert(AVLNode node, String key, Entry entry) {
        if (node == null) {
            return new AVLNode(key, entry);
        }
        if (key.compareTo(node.key) < 0) {
            node.left = insert(node.left, key, entry);
        } else if (key.compareTo(node.key) > 0) {
            node.right = insert(node.right, key, entry);
        } else {
            node.entries.add(entry);
            return node;
        }
        node.height = 1 + Math.max(getHeight(node.left),
getHeight(node.right));
        int balance = getBalance(node);
        if (balance > 1 && key.compareTo(node.left.key) < 0) {
            return rotateRight(node);
        }
        if (balance < -1 && key.compareTo(node.right.key) > 0) {
            return rotateLeft(node);
        }
        if (balance > 1 && key.compareTo(node.left.key) > 0) {
            node.left = rotateLeft(node.left);
            return rotateRight(node);
        }
```

```
        if (balance < -1 && key.compareTo(node.right.key) < 0) {
            node.right = rotateRight(node.right);
            return rotateLeft(node);
        }
        return node;
    }

    private AVLNode rotateLeft(AVLNode z) {
        AVLNode y = z.right;
        AVLNode T2 = y.left;
        y.left = z;
        z.right = T2;
        z.height = Math.max(getHeight(z.left), getHeight(z.right)) + 1;
        y.height = Math.max(getHeight(y.left), getHeight(y.right)) + 1;
        return y;
    }

    private AVLNode rotateRight(AVLNode z) {
        AVLNode y = z.left;
        AVLNode T3 = y.right;
        y.right = z;
        z.left = T3;
        z.height = Math.max(getHeight(z.left), getHeight(z.right)) + 1;
        y.height = Math.max(getHeight(y.left), getHeight(y.right)) + 1;
        return y;
    }

    private int getHeight(AVLNode node) {
        if (node == null) {
            return 0;
        }
        return node.height;
    }

    private int getBalance(AVLNode node) {
        if (node == null) {
            return 0;
        }
        return getHeight(node.left) - getHeight(node.right);
    }
```

Παρατηρήσεις:

Αυτό το τμήμα κώδικα υλοποιεί τις λειτουργίες εισαγωγής και περιστροφής για το δέντρο AVL, ένα δυαδικό δέντρο αναζήτησης που εξασφαλίζει ισορροπία σε κάθε εισαγωγή. Η μέθοδος εισαγωγής επιλέγει αρχικά τη σωστή θέση για το νέο κλειδί στο δέντρο και στη συνέχεια εισάγει τον κόμβο. Αν το κλειδί υπάρχει ήδη, η καταχώρηση των δεδομένων προσαρτάται στη λίστα entries του αντίστοιχου κόμβου. Κατά την εισαγωγή, ελέγχεται η

ισορροπία του δέντρου και, αν απαιτείται, γίνονται περιστροφές για να ανακτηθεί η ισορροπία. Οι περιστροφές (rotateLeft και rotateRight) εφαρμόζονται για τη διόρθωση της ισορροπίας του δέντρου. Το ύψος του κόμβου και η ισορροπία υπολογίζονται και ενημερώνονται μετά από κάθε εισαγωγή ή περιστροφή χρησιμοποιώντας τις μεθόδους getHeight και getBalance. Τέλος, η μέθοδος επιστρέφει τον ανανεωμένο κόμβο.

Search method for Count_of_Births

```
    public int search(AVLNode node, String region, int period) {
        if (node == null) {
            return -1;
        }
        if (region.compareTo(node.key) < 0) {
            return search(node.left, region, period);
        } else if (region.compareTo(node.key) > 0) {
            return search(node.right, region, period);
        } else {
            for (Entry entry : node.entries) {
                if (entry.getPeriod() == period &&
entry.getBirthDeath().equals("Births")) {
                    return entry.getCount();
                }
            }
            return -1;
        }
    }
```

Παρατηρήσεις:

Αυτό το τμήμα κώδικα αναλαμβάνει να αναζητήσει μια περιοχή σε ένα δέντρο AVL. Αν η περιοχή βρεθεί, ελέγχει τις καταχωρήσεις στη λίστα entries για την επιθυμητή περίοδο. Αν βρεθεί μια καταχώρηση με την επιθυμητή περίοδο και το γεγονός είναι γεννήσεις ("Births"), επιστρέφει τον αριθμό γεννήσεων για αυτήν την περιοχή και περίοδο. Αν δεν βρεθεί καταχώρηση για την περίοδο ή την περιοχή, επιστρέφει -1 για να υποδείξει ότι το αποτέλεσμα δεν βρέθηκε.

Modify Count of Births

```
    public boolean modify(AVLNode node, String region, int period, int
newCount) {
        if (node == null) {
            return false;
        }
        if (region.compareTo(node.key) < 0) {
            return modify(node.left, region, period, newCount);
        } else if (region.compareTo(node.key) > 0) {
```

```
            return modify(node.right, region, period, newCount);
        } else {
            for (Entry entry : node.entries) {
                if (entry.getPeriod() == period &&
entry.getBirthDeath().equals("Births")) {
                    entry.setCount(newCount);
                    return true;
                }
            }
            return false;
        }
    }
```

Παρατηρήσεις:

Αυτό το τμήμα κώδικα υλοποιεί τη λειτουργία τροποποίησης καταχώρησης σε ένα δέντρο AVL. Αναζητά πρώτα την περιοχή στο δέντρο. Αν βρει την περιοχή, ελέγχει τις καταχωρήσεις στη λίστα entries για την επιθυμητή περίοδο. Αν βρει μια καταχώρηση με την επιθυμητή περίοδο και το γεγονός είναι γεννήσεις , τότε τροποποιεί τον αριθμό γεννήσεων με τη νέα τιμή και επιστρέφει true. Αν δεν βρει καταχώρηση για την περίοδο ή την περιοχή, επιστρέφει false για να υποδείξει ότι η τροποποίηση δεν ήταν δυνατή.

Delete Regions

```
    public AVLNode delete(AVLNode node, String region, int period) {
        if (node == null) {
            return node;
        }
        if (region.compareTo(node.key) < 0) {
            node.left = delete(node.left, region, period);
        } else if (region.compareTo(node.key) > 0) {
            node.right = delete(node.right, region, period);
        } else {
            node.entries.removeIf(entry -> entry.getPeriod() == period
&& entry.getBirthDeath().equals("Births"));
            if (node.entries.isEmpty()) {
                if (node.left == null || node.right == null) {
                    node = (node.left != null) ? node.left :
node.right;
                } else {
                    AVLNode temp = minValueNode(node.right);
                    node.key = temp.key;
                    node.entries = temp.entries;
                    node.right = delete(node.right, temp.key, period);
                }
            }
        }
```

```
        if (node == null) {
            return node;
        }
        node.height = Math.max(getHeight(node.left),
getHeight(node.right)) + 1;
        int balance = getBalance(node);
        if (balance > 1 && getBalance(node.left) >= 0) {
            return rotateRight(node);
        }
        if (balance > 1 && getBalance(node.left) < 0) {
            node.left = rotateLeft(node.left);
            return rotateRight(node);
        }
        if (balance < -1 && getBalance(node.right) <= 0) {
            return rotateLeft(node);
        }
        if (balance < -1 && getBalance(node.right) > 0) {
            node.right = rotateRight(node.right);
            return rotateLeft(node);
        }
        return node;
    }

    private AVLNode minValueNode(AVLNode node) {
        AVLNode current = node;
        while (current.left != null) {
            current = current.left;
        }
        return current;
    }
```

Παρατηρήσεις:

Αυτή η μέθοδος διαγραφής σε ένα δέντρο AVL αφαιρεί μια καταχώρηση που αντιστοιχεί σε μια περιοχή και περίοδο. Ξεκινάει αναζητώντας την περιοχή στο δέντρο. Αν βρει την περιοχή, αφαιρεί την καταχώρηση αντίστοιχα με την περίοδο που δόθηκε, εάν υπάρχει. Αν η λίστα καταχωρήσεων για τον κόμβο αυτόν γίνει άδεια μετά τη διαγραφή, τότε ο κόμβος διαγράφεται. Στη συνέχεια, ελέγχει την ισορροπία του δέντρου και εφαρμόζει τις κατάλληλες περιστροφές για να διατηρήσει την ισορροπία του. Τέλος, επιστρέφει τον ριζικό κόμβο του δέντρου μετά τη διαγραφή.

AVL με Count_of_Births

Search for min Birth_Count Regions

```
    public AVLNode findMin(AVLNode node) {
```

```
        if (node == null) {
            return null;
        }
        while (node.left != null) {
            node = node.left;
        }
        return node;
    }
```

Search for max Birth_Count Regions

```
    public AVLNode findMax(AVLNode node) {
        if (node == null) {
            return null;
        }
        while (node.right != null) {
            node = node.right;
        }
        return node;
    }
```

Παρατηρήσεις:

Οι μέθοδοι findMin και findMax χρησιμοποιούνται για την εύρεση του ελάχιστου και μέγιστου κόμβου αντίστοιχα σε ένα δέντρο AVL. Αρχίζουν από τον ριζικό κόμβο και στη συνέχεια διατρέχουν το δέντρο προς τα αριστερά ή τα δεξιά αντίστοιχα μέχρι να βρουν έναν κόμβο χωρίς αριστερό ή δεξί παιδί, αντίστοιχα, ο οποίος είναι ο ελάχιστος ή ο μέγιστος κόμβος στο δέντρο. Στη συνέχεια επιστρέφουν τον αντίστοιχο κόμβο. Αν ο κόμβος είναι άδειος, επιστρέφουν null.

Hashing with chains

HashTableWithChaining

```
    public HashTableWithChaining(int size) {
        this.size = size;
        table = new ArrayList<>(size);
        for (int i = 0; i < size; i++) {
            table.add(new LinkedList<>());
        }
    }

    private int hash(String region) {
        int hashValue = 0;
        for (char c : region.toCharArray()) {
            hashValue += (int) c;
```

```
        }
        return hashValue % 3;
    }
```

Παρατηρήσεις:

Η μέθοδος HashTableWithChaining είναι ο κατασκευαστής της κλάσης HashTableWithChaining και δημιουργεί έναν πίνακα κατακερματισμού με αλυσίδες. Αρχικοποιεί το μέγεθος του πίνακα στο μέγεθος που δίνεται ως παράμετρος και δημιουργεί μια λίστα από λίστες, όπου κάθε λίστα αντιπροσωπεύει την αλυσίδα στο συγκεκριμένο κουτί του πίνακα. Η μέθοδος hash χρησιμοποιείται για τον υπολογισμό του κατακερματισμού μιας περιοχής. Προσθέτει την ASCII τιμή κάθε χαρακτήρα της περιοχής και στη συνέχεια επιστρέφει το υπόλοιπο της διαίρεσης του αθροίσματος με το μέγεθος του πίνακα, προκειμένου να επιστρέψει έναν δείκτη στον πίνακα.

Insert/Search/Modify/Delete

```
    public void insert(Entry entry) {
        int index = hash(entry.getRegion());
        table.get(index).add(entry);
    }

    public Entry search(String region, int period) {
        int index = hash(region);
        for (Entry entry : table.get(index)) {
            if (entry.getRegion().equals(region) && entry.getPeriod()
== period) {
                return entry;
            }
        }
        return null;
    }

    public boolean modify(String region, int period, int newCount) {
        Entry entry = search(region, period);
        if (entry != null) {
            entry.setCount(newCount);
            return true;
        }
        return false;
    }

    public boolean delete(String region, int period) {
        int index = hash(region);
        LinkedList<Entry> chain = table.get(index);
        for (Entry entry : chain) {
```

```
            if (entry.getRegion().equals(region) && entry.getPeriod()
== period) {
                chain.remove(entry);
                return true;
            }
        }
        return false;
    }
```

Παρατηρήσεις:

Η μέθοδος insert εισάγει ένα στοιχείο στον πίνακα κατακερματισμού. Υπολογίζει τον δείκτη (index) χρησιμοποιώντας τη μέθοδο hash, και στη συνέχεια προσθέτει το στοιχείο στην αλυσίδα που αντιστοιχεί στον συγκεκριμένο δείκτη. Η μέθοδος search αναζητά ένα στοιχείο με βάση την περιοχή και την περίοδο. Υπολογίζει τον δείκτη χρησιμοποιώντας τη μέθοδο hash και στη συνέχεια διατρέχει την αλυσίδα στον συγκεκριμένο δείκτη, αναζητώντας το στοιχείο που ταιριάζει με την περιοχή και την περίοδο. Η μέθοδος modify τροποποιεί τον αριθμό εγγραφών για μια συγκεκριμένη περιοχή και περίοδο. Εκτελεί αναζήτηση χρησιμοποιώντας τη μέθοδο search, και αν βρεθεί το αντίστοιχο στοιχείο, το τροποποιεί με τη νέα τιμή. Η μέθοδος delete διαγράφει ένα στοιχείο από τον πίνακα κατακερματισμού. Υπολογίζει τον δείκτη χρησιμοποιώντας τη μέθοδο hash, και στη συνέχεια διατρέχει την αλυσίδα στον συγκεκριμένο δείκτη, αναζητώντας και διαγράφοντας το στοιχείο που ταιριάζει με την περιοχή και την περίοδο.

Unified Project

Main menu

```
    static void avlMenu(Scanner scanner) {
        while (true) {
            System.out.println("AVL Tree Menu:");
            System.out.println("1. Region-based AVL tree (Project 1)");
            System.out.println("2. Count-based AVL tree (Project 2)");
            System.out.println("3. Back to main menu");
            System.out.print("Enter your choice: ");
            int choice = scanner.nextInt();
            scanner.nextLine();

            if (choice == 1) {
                Project1.mainMenu();
            } else if (choice == 2) {
                Project2.mainMenu();
            } else if (choice == 3) {
                mainMenu(scanner);
            } else {
```

```
                System.out.println("Invalid choice. Please try
again.");
            }
        }
    }
```

Project1 Menu

```
public static void mainMenu() {
        AVLTree tree = new AVLTree();
        List<Entry> dataList = Main.readFromFile("data.txt");

        for (Entry entry : dataList) {
            tree.root = tree.insert(tree.root, entry.getRegion(),
entry);
        }

        Scanner scanner = new Scanner(System.in);

        while (true) {
            System.out.println("Project 1 Menu:");
            System.out.println("1. Print AVL tree");
            System.out.println("2. Search for count of births");
            System.out.println("3. Modify the amount of births");
            System.out.println("4. Delete data for a region and
period");
            System.out.println("5. Exit to AVL menu");
            System.out.print("Enter your choice: ");
            int choice = scanner.nextInt();
            scanner.nextLine();

            if (choice == 1) {
                System.out.println("Preorder traversal of the
constructed AVL tree is:");
                tree.preOrder(tree.root, 0);
            } else if (choice == 2) {
                System.out.print("Enter the region: ");
                String region = scanner.nextLine();
                System.out.print("Enter the period: ");
                int period = scanner.nextInt();
                int count = tree.search(tree.root, region, period);
                if (count != -1) {
                    System.out.println("Count of births for " + region
+ " in period " + period + ": " + count);
                } else {
                    System.out.println("No data found for the specified
region and period.");
                }
```

```
            } else if (choice == 3) {
                System.out.print("Enter the region: ");
                String region = scanner.nextLine();
                System.out.print("Enter the period: ");
                int period = scanner.nextInt();
                System.out.print("Enter the new count of births: ");
                int newCount = scanner.nextInt();
                boolean success = tree.modify(tree.root, region,
period, newCount);
                if (success) {
                    System.out.println("Updated count of births for " +
region + " in period " + period + " to " + newCount);
                } else {
                    System.out.println("No data found for the specified
region and period.");
                }
            } else if (choice == 4) {
                System.out.print("Enter the region: ");
                String region = scanner.nextLine();
                System.out.print("Enter the period: ");
                int period = scanner.nextInt();
                tree.root = tree.delete(tree.root, region, period);
                System.out.println("Deleted data for " + region + " in
period " + period);
            } else if (choice == 5) {
                System.out.println("Returning to AVL menu...");
                UnifiedProject.avlMenu(scanner);
                break;
            } else {
                System.out.println("Invalid choice. Please try
again.");
            }
        }
    }
```

Project 2 Menu

```
class Project2 {
    public static void mainMenu() {
        List<Entry> dataList = Main.readFromFile("data.txt");
        AVLTree tree = new AVLTree();

        for (Entry entry : dataList) {
            if (entry.getBirthDeath().equals("Births")) {
                tree.root = tree.insert(tree.root, entry.getCount(),
entry);
            }
        }
```

```
        Scanner scanner = new Scanner(System.in);
        while (true) {
            System.out.println("Project 2 Menu:");
            System.out.println("1. Search for the region(s) with the
minimum birth count");
            System.out.println("2. Search for the region(s) with the
maximum birth count");
            System.out.println("3. Exit to AVL menu");
            System.out.print("Enter your choice: ");
            int choice = scanner.nextInt();
            scanner.nextLine();

            if (choice == 1) {
                AVLTree.AVLNode minNode = tree.findMin(tree.root);
                if (minNode != null) {
                    System.out.println("Region(s) with the minimum
birth count (" + minNode.countKey + "):");
                    for (Entry entry : minNode.entries) {
                        System.out.println("Region: " +
entry.getRegion() + ", Period: " + entry.getPeriod());
                    }
                } else {
                    System.out.println("Tree is empty.");
                }
            } else if (choice == 2) {
                AVLTree.AVLNode maxNode = tree.findMax(tree.root);
                if (maxNode != null) {
                    System.out.println("Region(s) with the maximum
birth count (" + maxNode.countKey + "):");
                    for (Entry entry : maxNode.entries) {
                        System.out.println("Region: " +
entry.getRegion() + ", Period: " + entry.getPeriod());
                    }
                } else {
                    System.out.println("Tree is empty.");
                }
            } else if (choice == 3) {
                System.out.println("Returning to AVL menu...");
                UnifiedProject.avlMenu(scanner);
                break;
            } else {
                System.out.println("Invalid choice. Please try
again.");
            }
        }
    }
}
```

Project 3 Menu

```
class Project3 {
    public static void mainMenu() {
        List<Entry> dataList = Main.readFromFile("data.txt");

        HashTableWithChaining hashTable = new HashTableWithChaining(3);

        for (Entry entry : dataList) {
            if (entry.getBirthDeath().equals("Births")) {
                hashTable.insert(entry);
            }
        }

        Scanner scanner = new Scanner(System.in);
        while (true) {
            System.out.println("Project 3 Menu:");
            System.out.println("1. Search for the amount of birth in a
period and a region");
            System.out.println("2. Modify the count of birth in a
period and a region");
            System.out.println("3. Delete an index when it comes to a
region");
            System.out.println("4. Exit to main menu");
            System.out.print("Enter your choice: ");
            int choice = scanner.nextInt();
            scanner.nextLine();

            if (choice == 1) {
                System.out.print("Enter region: ");
                String region = scanner.nextLine();
                System.out.print("Enter period: ");
                int period = scanner.nextInt();
                Entry result = hashTable.search(region, period);
                if (result != null) {
                    System.out.println("Region: " + result.getRegion()
+ ", Period: " + result.getPeriod() + ", Births: " +
result.getCount());
                } else {
                    System.out.println("Entry not found.");
                }
            } else if (choice == 2) {
                System.out.print("Enter region: ");
                String region = scanner.nextLine();
                System.out.print("Enter period: ");
                int period = scanner.nextInt();
                System.out.print("Enter new count of births: ");
                int newCount = scanner.nextInt();
```

```
                boolean success = hashTable.modify(region, period,
newCount);
                if (success) {
                    System.out.println("Entry updated successfully.");
                } else {
                    System.out.println("Entry not found.");
                }
            } else if (choice == 3) {
                System.out.print("Enter region: ");
                String region = scanner.nextLine();
                System.out.print("Enter period: ");
                int period = scanner.nextInt();
                boolean success = hashTable.delete(region, period);
                if (success) {
                    System.out.println("Entry deleted successfully.");
                } else {
                    System.out.println("Entry not found.");
                }
            } else if (choice == 4) {
                    UnifiedProject.mainMenu(scanner);
                    break;
            } else {
                System.out.println("Invalid choice. Please try
again.");
            }
        }
    }
}
```

Σημειώσεις 1-2

Σημείωση 1

Σκεφτείτε να οργανώσετε αποδοτικά την πληροφορία (λίστα ή πίνακα) που αντιστοιχεί στον κάθε κόμβο του ΔΔΑ καθώς και την αλυσίδα του Hash Table ως προς το πεδίο Period. Τι θα κάνατε;

Όλα τα periods είναι αποθηκευμένα σε μια λίστα και όπως όλα τα άλλα μέλη της λίστας μπορούν να καλεστούν οποιαδήποτε στιγμή με την συνάρτηση getperiod() με σκοπό να γίνουν τυχών τροποποιήσεις ή διαγραφές.

Σημείωση 2

Σκεφτείτε το δέντρο αναζήτησης να είναι ζυγισμένο (AVL, ή red-black ή (α,β) δέντρα). Τι θα κάνατε;

Εξετάζω κάθε φορά την ισοζύγιση της ρίζας κάθε υποδέντρου και σε περίπτωση που δεν έχει τις επιτρεπόμενες τιμές -1,0,1 εφαρμόζω μια αλληλουχία η και όχι περιστροφών ανάλογα με την συνθήκη στην οποία βρίσκεται την εκάστοτε φορά. Συνεχίζω με τον ίδιο τρόπο μέχρι όλοι οι κόμβοι να έχουν τις επιτρεπόμενες τιμές και άρα δημιουργώ ένα AVL tree.