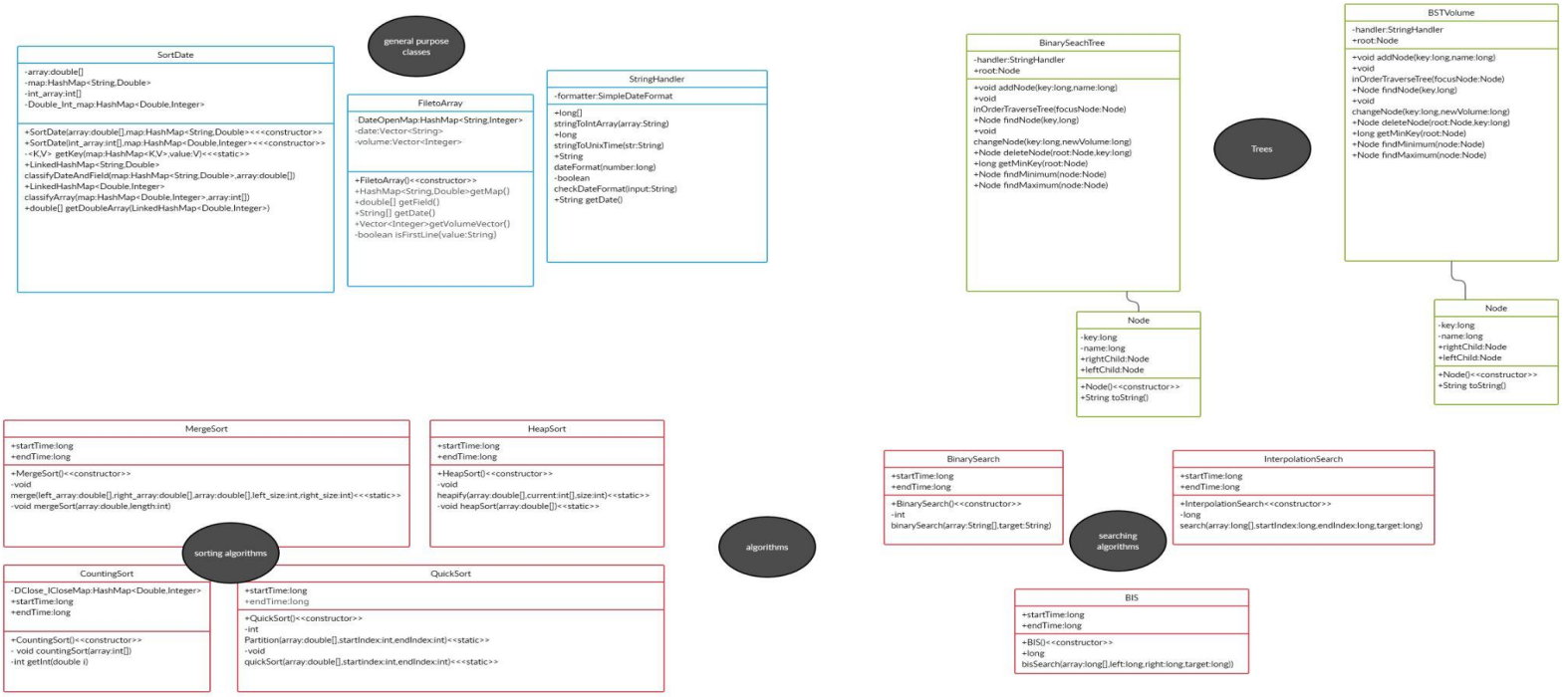


Project in Data Structures

Στοιχεια ομαδας:

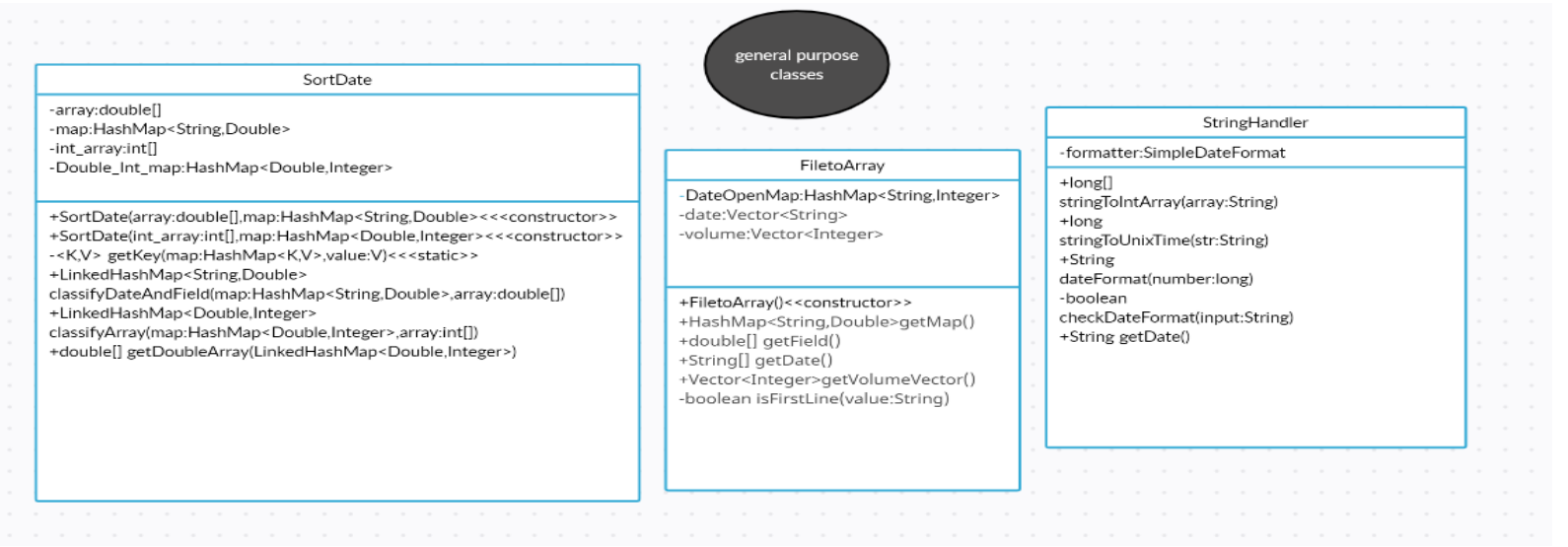
Όνομα: Διονυσία **Επώνυμο:** Ψυρρή **ΑΜ:** 1080424

Όνομα: Χρήστος **Επώνυμο:** Έρδας **ΑΜ:** 1072543

Περιγραφή μεθοδολογίας: Όπως φαίνεται και απο το uml διάγραμμα που κατασκευάσαμε, “χωρίσαμε” το project με βάση τρία είδη:

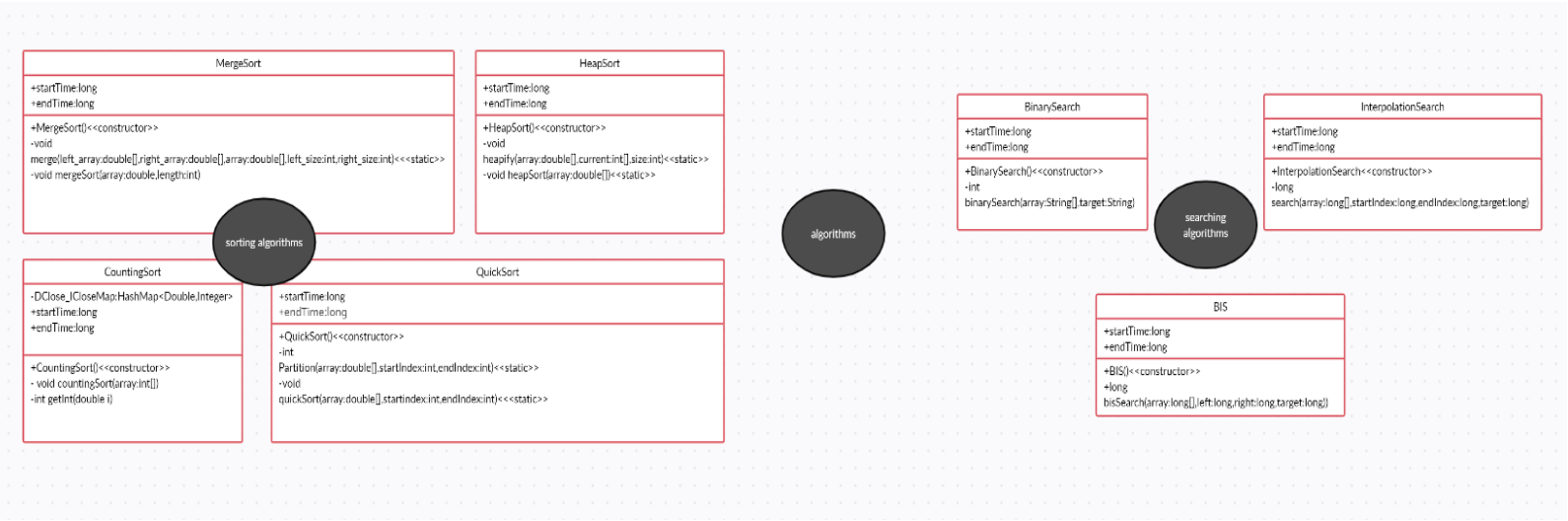
1. Κλάσεις γενικού τύπου

Αντικείμενα αυτών των κλάσεων χρησιμοποιούνται σε όλες σχεδόν τις άλλες κλάσεις, στόχος αυτών είναι να “πάρουν” τα στοιχεία από τα αρχεία και να διαθέτουν μεθόδους οι οποίες επιτρέπουν τις κατάλληλες μετατροπές που απαιτούνται για την υλοποίηση των αλγορίθμων και των δέντρων.

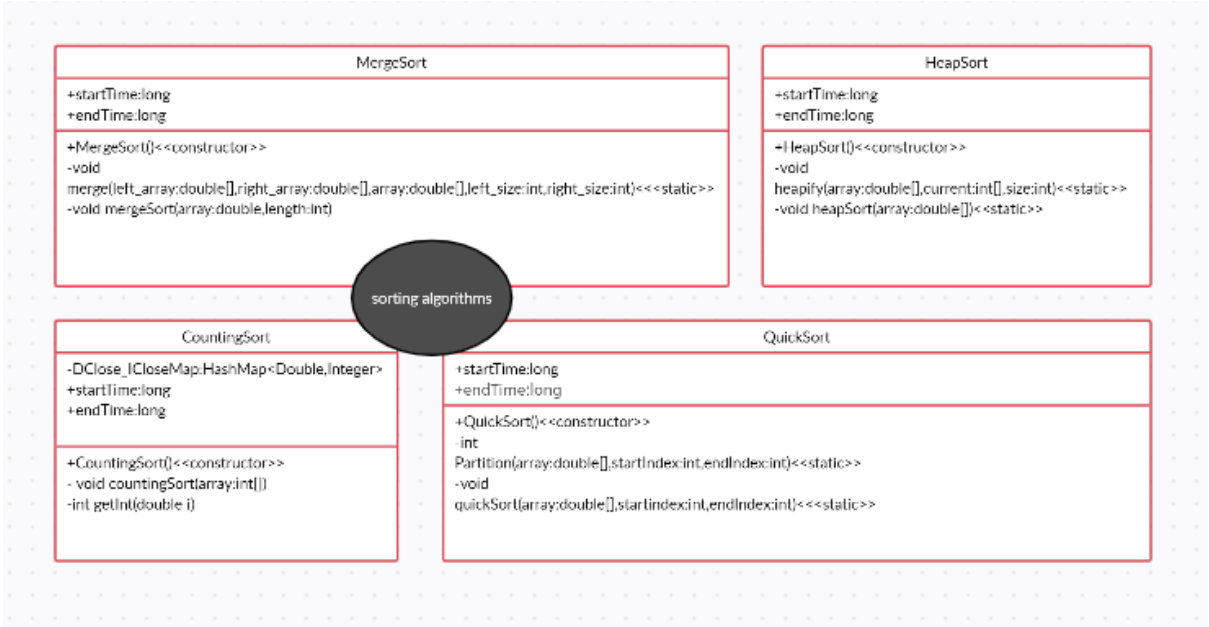


2. Αλγόριθμοι

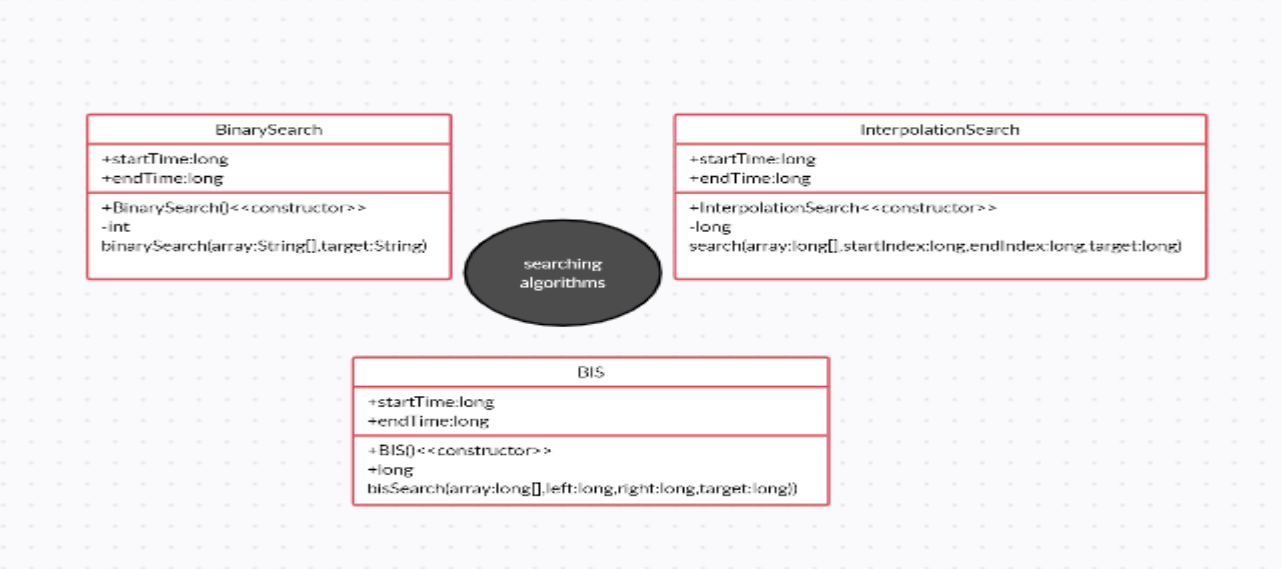
Οι αλγόριθμοι είναι κλάσεις που υλοποιούν τα ζητούμενα του πρώτου μέρους του project,και χωρίζονται σε δύο κατηγορίες:



α)Sorting algorithms

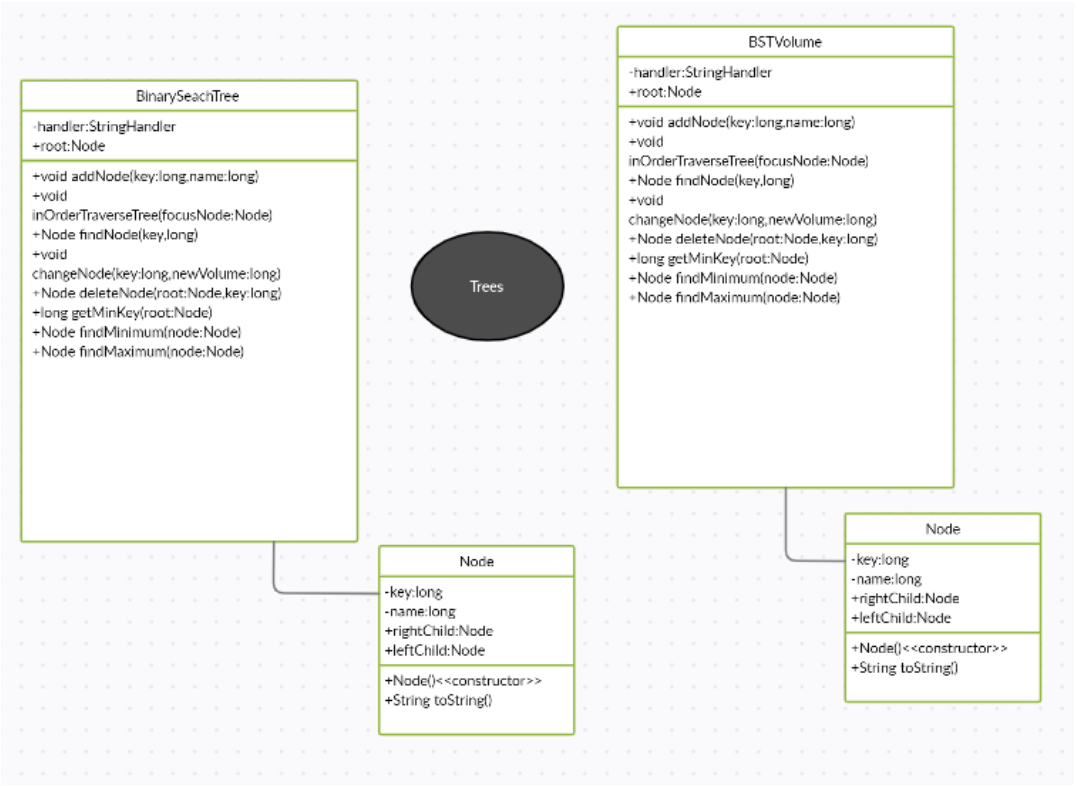


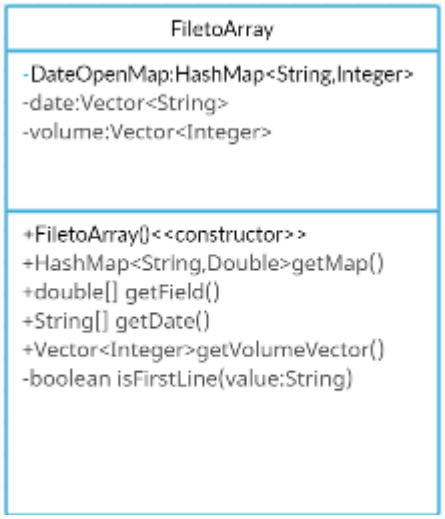
β)Searching algorithms



3.Δεντρα

Τα δέντρα μαζί με το hashing αποτελούν το δεύτερο μέρος του project,και όλα αυτά αναπαριστώνται απο το Menu.


BinarySeachTree
-handler:StringHandler
+root:Node
+void addNode(key:long,name:long)
+void
inOrderTraverseTree(focusNode:Node)
+Node findNode(key,long)
+void
changeNode(key:long,newVolume:long)
+Node deleteNode(root:Node,key:long)
+long getMinKey(root:Node)
+Node findMinimum(node:Node)
+Node findMaximum(node:Node)
Trees
BSTVolume
-handler:StringHandler
+root:Node
+void addNode(key:long,name:long)
+void
inOrderTraverseTree(focusNode:Node)
+Node findNode(key,long)
+void
changeNode(key:long,newVolume:long)
+Node deleteNode(root:Node,key:long)
+long getMinKey(root:Node)
+Node findMinimum(node:Node)
+Node findMaximum(node:Node)
Node
-key:long
-name:long
+rightChild:Node
+leftChild:Node
+Node()<<constructor>>
+String toString()
Node
-key:long
-name:long
+rightChild:Node
+leftChild:Node
+Node()<<constructor>>
+String toString()



FiletoArray:

Η συγκεκριμένη κλάση χρησιμοποιείται για να διαβάζει τα αρχεία που έχουν δοθεί. Γι αυτό το λόγο διαθέτει έναν constructor που παίρνει σαν ορίσματα δύο Strings.Το πρώτο δείχνει το όνομα του αρχείου που θα διαβάσει(μ αυτό τον τρόπο για να διαβάσουμε διαφορετικό αρχείο απλά δίνουμε διαφορετικό πρωτο όρισμα), ενώ το δεύτερο όρισμα δηλώνει το δεύτερο πεδίο του αρχείου(το πρώτο είναι πάντα το date). Έπειτα κατασκευάζουμε object BufferedReader το οποίο διαβάζει το αρχείο με την εντολη read().Αποθηκεύουμε τις τιμές που διαβάζει χωρίζοντας ‘τες ταυτόχρονα με βάση το “,”(κομμα) σε ενα array τύπου String(types).Αναλυτικότερα:

```
while((readline = reader.readLine()) != null){
                String [] types = readline.split("[,]");



                if(isFirstLine(types[1])){

                    for(int i = 0; i<types.length; ++ i){
                        if(types[i].equals(wordToAvoid))
                            numberOfType = i;
                    }

                }
                else{
                    if(wordToAvoid == "Volume"){
                        date.add(types[0]);
                        volume.add(Integer.parseInt(types[numberOfType]));

                    }
                    else{
```

```
DateOpenMap.put(types[0],Double.parseDouble(types[numberOfType]));
                    }
}
```

Επειδή το αρχείο διαθέτει τα ονόματα των πεδίων(Date,Open…)τα οποία δεν θέλουμε να βρίσκονται σε δομές, φτιάξαμε μια μέθοδος για να ελέγξουμε την πρώτη γραμμή την οποία δεν θα προσθέσουμε στους πίνακες και στα maps.

```
private boolean isFirstLine(String value){
        try{
            Double.parseDouble(value);
            return false;
        }
        catch(NumberFormatException e){
            return true;
        }
}
```

Η isFirstLine δεχεται ενα όρισμα τυπου String το οποίο όταν την καλέσουμε θα κοιτάει το πεδίο Open το οποίο περιέχει Doubles εκτός από την πρώτη γραμμή ετσι η isFirstLine προσπαθεί να μετατρέψει το String σε Double, το οποίο αν είναι η πρώτη γραμμή δηλαδή η λέξη Open δεν μπορεί να το μετατρέψει και επιστρέφει true.Αφού αποφύγουμε αυτο προσθέτουμε τις τιμές του Open(ή του Volume αν το δεύτερο όρισμα ειναι “Volume”) σε κατάλληλες δομές και δημιουργούμε μεθόδους που επιτρέπουν την πρόσβαση αυτών από άλλες κλάσεις:

```
    public HashMap<String,Double> getMap(){
        return DateOpenMap;
    }

    public double[] getField(){

        Vector<Double> vec = new Vector<Double>();
        double[] array;

        for(Map.Entry<String,Double> mapElement : DateOpenMap.entrySet()){
            vec.add((double)mapElement.getValue());
        }

        array = new double[vec.size()];

        for(int i=0;i<vec.size();i++){
            array[i] = vec.get(i);
        }
        return array;
    }
```

```
public String[] getDate(){
    String[] array = date.toArray(new String[date.size()]);
    return array;
}


public Vector<Integer> getVolumeVector(){
    return volume;
}
```

SortDate:

Η κλάση αυτή δημιουργήθηκε επειδή με τους αλγορίθμους sort ταξινομούμε τις τιμές του πεδίου Open ή Volume, μετά την ταξινόμηση πρέπει να αντιστοιχίσουμε και τις τιμές Date έτσι ώστε να ταξινομηθούν και αυτές με βάση τις παραπάνω(Open και Volume).Επειδή χρησιμοποιούμε maps και οι sort επιστρέφουν arrays φτιάξαμε την μέθοδο **classifyDateAndField:**

```
public LinkedHashMap<String,Double> classifyDateAndField(HashMap<String,Double>map,double[] array){

        LinkedHashMap<String,Double> sortedMap = new LinkedHashMap<String,Double>();

        for(double number : array){
            sortedMap.put(getKey(map,number),number);
        }
        return sortedMap;
}
```

Η **classifyDateAndField** δεχεται σαν όρισμα το αρχικό map και το ταξινομημένο array π επιστρέφουν οι sort αλγόριθμοι και ταξινομεί τα dates με βαση αυτο.

Για να γίνει αυτο χρειαζόμαστε το key του map(πεδίο Date) το οποίο επειδή δεν μπορούμε να το πάρουμε κατευθείαν απο έτοιμη μέθοδο της java φτιάξαμε την μέθοδο **getKey** η οποία κάνει χρήση Generic <K,V>(αντιστοιχεί σε key και value του map):

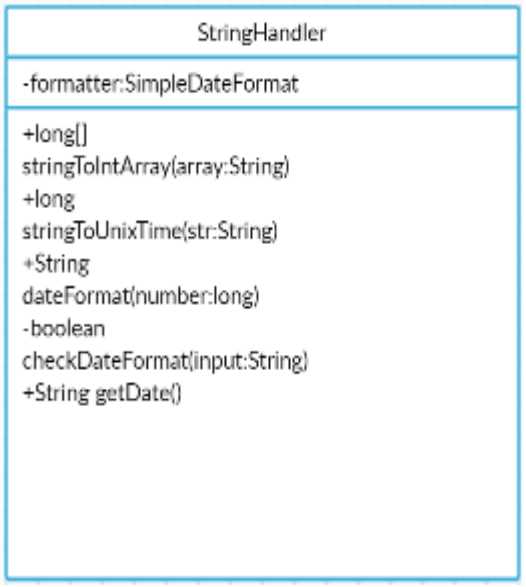
```
private static <K, V> K getKey(HashMap<K, V> map, V value) {

        for (K key: map.keySet()){
            if (value.equals(map.get(key))){
                return key;
            }
        }
        return null;
}
```

Ουσιαστικά με ένα **enhanced for loop** διατρέχουμε το map και αν το value που έχουμε δώσει σαν δεύτερο όρισμα υπάρχει μας επιστρέφει το Key το οποίο είναι τύπου K, αλλιώς επιστρέφει null(στην συγκεκριμένη περίπτωση δεν θα επιστρέψει ποτε null καθώς βάζουμε τιμές απο το sorted array που υπήρχαν απο πριν.

Τέλος επειδή στον counting sort ναι μεν ζητείται ταξινόμηση του πεδίου date με βάση του Open(που είναι double) πρέπει να γίνει στρογγυλοποίηση στον πλησιέστερο integer έτσι η **classifyDateAndField** δεν θα έχει πλέον doubles φτιάξαμε τις μεθόδους **getDoubleArray** και **classifyArray** οι οποίες θα ταξινομήσουν τους integers στα αντίστοιχα doubles.

```
public double[] getDoubleArray(LinkedHashMap<Double,Integer> map){


        Vector<Double> vec = new Vector<Double>();
        double[] array;

        for(Map.Entry<Double,Integer> mapElement : map.entrySet()){
            vec.add((double)mapElement.getKey());
        }

        array = new double[vec.size()];

        for(int i=0;i<vec.size();i++){
            array[i] = vec.get(i);
        }
        return array;
}
```

```
public LinkedHashMap<Double,Integer> classifyArray(HashMap<Double,Integer> map, int[] array){

        LinkedHashMap<Double,Integer> sortedMap = new LinkedHashMap<Double,Integer>();

        for(int number : array){
            sortedMap.put(getKey(map,number),number);
        }

        return sortedMap;
}
```



StringHandler:

Η κλάση αυτή δημιουργήθηκε για χειρίζεται τα τύπου String δεδομένα που παίρνουμε από το πεδίο dates. Πιο συγκεκριμένα στους search αλγορίθμους που ζητείται να υλοποιήσουμε πρέπει ο χρήστης να εισάγει τιμή μέσω του πληκτρολογίου, γι αυτη την τιμή ωστόσο πρέπει να είμαστε σίγουροι ότι θα είναι της μορφής yyyy-mm-dd και επομένως δημιουργούμε τις μεθόδους **checkDateFormat** και **getDate** οι οποίες μας επιτρέπουν αρχικά(με την **checkDateFormat**) να ελέγξουμε μέσω ενός regular expression αν η τιμή που έδωσε ο χρήστης ακολουθεί το πρότυπο(yyyy-mm-dd)και έπειτα με την **getDate** να κάνει χρήση αμυντικού προγραμματισμού μέχρι ο χρήστης να δώσει την ημερομηνία στην κατάλληλη μορφή. Αναλυτικότερα:

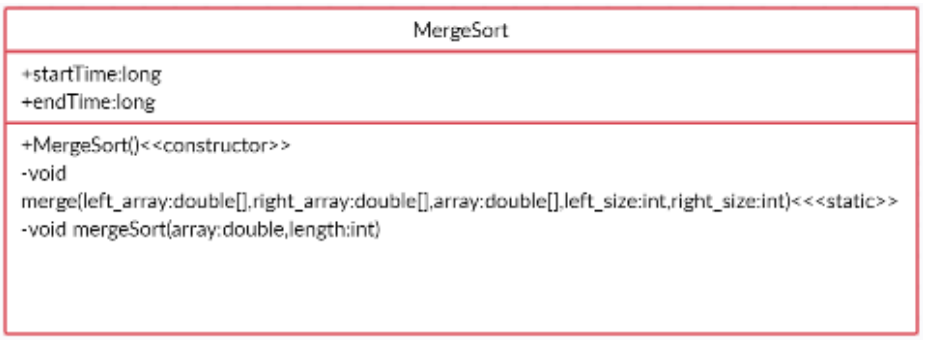
```
private boolean checkDateFormat(String input){

              Pattern pattern = Pattern.compile("^(19|20)\\d\\d[- /.](0[1-9]|1[012])[-
/.](0[1-9]|[12][0-9]|3[01])$");
        Matcher matcher = pattern.matcher(input);

        return matcher.matches();
    }


    public String getDate(){
        String stringInput;
        Scanner input = new Scanner(System.in);
        while(true){
            System.out.println("Enter a date (yyyy-mm-dd): ");
            System.out.print("> ");
```

```
        stringInput = input.nextLine();

        if(checkDateFormat(stringInput))
            break;
        else{
            System.out.print("\033[H\033[2J");
            System.out.flush();
            System.out.println("Please enter a valid date (yyyy-mm-dd)");
        }
    }
    return stringInput;
}
```

Επιπλέον επειδή κάποιοι search αλγόριθμοι(και δέντρα) απαιτούν την χρήση αριθμών πρέπει να μετατρέψουμε τα Strings σε αριθμούς με στόχο ωστόσο να αποτυπώνουν και την σωστή ημερομηνία.Γι αυτό χρησιμοποιούμε το unix timestamp το οποίο είναι αναπαράσταση χρόνου με βάση μια συγκεκριμένη ημερομηνία(1η Ιανουαρίου 1970) το οποίο δημιουργήθηκε από μια ομάδα προγραμματιστών τοτε. Επομένως κατασκευάσαμε μεθόδους που μετατρέπουν την ημερομηνία σε format yyyy-mm-dd (με την βοήθεια SimpleDateFormat) σε unix timestamp και αντιστρόφως(για να επαναφέρουμε το format οταν χρειαζεται).

```
    public long[] stringToIntArray(String[] array) {

        long[] int_array = new long[array.length];

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

            int_array[i] = stringToUnixTime(array[i]);
        }
        return int_array;
    }


    public long stringToUnixTime(String str) {

        try{
            Date date = formatter.parse(str);
            return date.getTime();
        }
        catch(ParseException e){
```

```
            return -1;
        }
    }

    public String dateFormat(long number){
        return formatter.format(number);
    }
```



MergeSort:

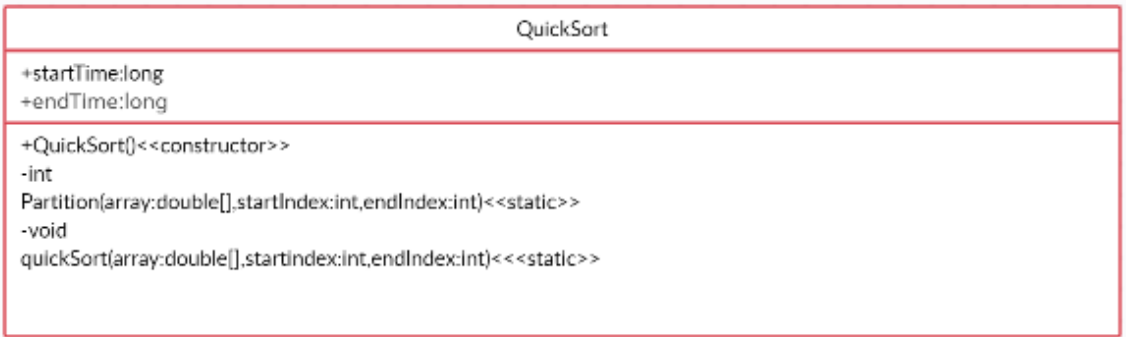
Για το πρώτο ερώτημα ζητήθηκε υλοποίηση των αλγορίθμων **Merge Sort** και **Quick Sort** με βάση το πεδίο Open. Το πεδίο Open περιέχει αριθμούς double τους οποίους έπρεπε να ταξινομήσουμε με αύξουσα σειρά και έπειτα να αντιστοιχήσουμε τα αντίστοιχα Date. Και στους δύο αλγορίθμους φτιάξαμε constructor στους οποίους δημιουργήσαμε αντικείμενα **FiletoArray** και **SortDate** για τους λόγους που εξηγήσαμε και παραπάνω. Έπειτα οι μέθοδοι **mergeSort** και **merge** υλοποιούσαν τον αλγόριθμο Merge sort:

```
private static void merge(double[] left_array,double[] right_array, double[] array,int left_size, int right_size){

        int i=0; int left_array_counter=0; int right_array_counter=0;

        while(left_array_counter<left_size && right_array_counter<right_size){

            if(left_array[left_array_counter]<right_array[right_array_counter]){
                array[i++] = left_array[left_array_counter++];
            }
            else{
                array[i++] = right_array[right_array_counter++];
            }
        }
        //this while loops checks which part of right or left has not end
        while(left_array_counter<left_size){
```

```
        array[i++] = left_array[left_array_counter++];
    }
    while(right_array_counter<right_size){
    array[i++] = right_array[right_array_counter++];
    }
}

private void mergeSort(double array[], int lenght){

    if (lenght <= 1){
        endTime = System.nanoTime();
        return;
    } //checks if there is one element in the array

    int mid = lenght / 2;
    double  left_array[] = new double[mid];
    double  right_array[] = new double[lenght-mid];
    int k=0;

    for(int i = 0;i<lenght;++i){
        if(i<mid){
            left_array[i] = array[i];
        }
        else{
            right_array[k] = array[i];
            k = k+1;
        }
    }
    mergeSort(left_array,mid);
    mergeSort(right_array,lenght-mid);
    merge(left_array,right_array,array,mid,lenght-mid);
}
```

mergeSort(): Η mergeSort() χωρίζει τον πίνακα σε δυο υπο πίνακες τους οποίους γεμίζει μέσω ενός for loop.Έπειτα καλεί τον εαυτό της για τους δύο υπο πίνακες και συνεχίζει εως ότου κάθε στοιχείο να αποτελεί έναν πίνακα. Τέλος καλεί την merge() η οποία ταξινομεί τα στοιχεία κατά αύξουσα σειρά και τα τοποθετεί σε έναν πίνακα.

Χρονική πολυπλοκότητα **Merge Sort:** σε όλες τις περιπτώσεις O(nlogn).Με μέτρηση των millisecond που απαιτείται για να υλοποιηθεί ο αλγόριθμος(με χρήση **nanoTime()**πριν και μετά την υλοποίηση του): **1.2821 milliseconds.**



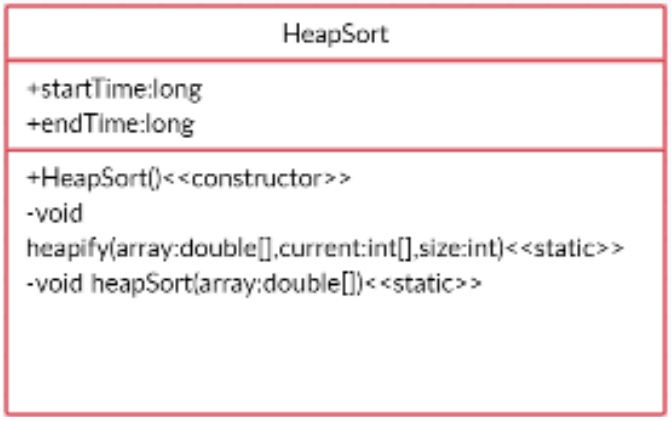
QuickSort:

Επιλέξαμε ένα στοιχείο ως pivot(εμείς βάλαμε το τελευταίο) και το αφαιρέσαμε από τον πίνακα εισόδου.Έπειτα το τοποθετήσαμε στην σωστή θέση του πίνακα και τοποθετήσαμε όλα τα μικρότερα στοιχεία από το pivot αριστερά και όλα τα μεγαλύτερα δεξιά. Ξεκινάμε από το αριστερότερο στοιχείο και παρακολουθούμε δείκτη μικρότερων (ή ίσων με) στοιχείων όπως i. Έπειτα ενώ διασχίζουμε, αν βρούμε ένα μικρότερο στοιχείο, ανταλλάσσουμε το τρέχον στοιχείο με array [i]. Διαφορετικά αγνοούμε το τρέχον στοιχείο.Αναλυτικότερα:

```
private static int Partition(double [] array, int startIndex,int endIndex){
  double pivot=array[endIndex]; //make the last element as pivot element
  int i = (startIndex-1);
    for (int j=startIndex; j<endIndex; j++) {
        // If current element is smaller than the pivot
        if (array[j] <= pivot)
        {
           // Increment index of smaller element
            i++;
            double temp = array[i];
            array[i] = array[j];
            array[j] = temp;
        }
    }

    double temp = array[i+1];
    array[i+1] = array[endIndex];
    array[endIndex] = temp;

    return i+1;

}
private static void quickSort(double [] array,int startIndex,int endIndex){
    if(startIndex<endIndex){
        int pIndex= Partition(array, startIndex, endIndex); //stores the position of pivot
        quickSort(array, startIndex, pIndex-1); //sorts the left side of pivot
        quickSort(array, pIndex+1, endIndex);   //sorts the right side of pivot
    }
}
```

Χρονική πολυπλοκότητα **Quick Sort:** Χειρότερης περίπτωσης: $O(n^2)$, μέσης καλύτερης περίπτωσης: O(nlogn). Με μέτρηση των millisecond που απαιτείται για να υλοποιηθεί ο αλγόριθμος(με χρήση **nanoTime()**πριν και μετά την υλοποίηση του): **4.5654 milliseconds.**

Σύγκριση **QuickSort** με **MergeSort:** Όπως φαίνεται και απο τους χρόνους σε millisecond που μετρήσαμε, η **mergeSort** είναι πιο αποτελεσματική και λειτουργεί πιο γρήγορα από την **quickSort** σε περίπτωση μεγαλύτερου μεγέθους πίνακα ή συνόλων δεδομένων όπως στην δική μας περίπτωση.



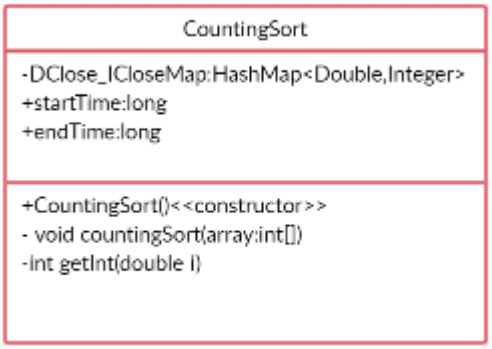
HeapSort:

Για το δεύτερο ερώτημα ζητήθηκε να υλοποιηθούν οι αλγόριθμοι **HeapSort** και **CountingSort** με βάση το πεδίο **close** (γι αυτο το δεύτερο όρισμα του constructor θα είναι “close”στο αντικείμενο **FiletoArray** που θα κατασκευάσουμε). το **Heap Sort** το υλοποιήσαμε χρησιμοποιώντας ένα μέγιστο σωρό. Το μέγιστο στοιχείο βρίσκεται στην ρίζα του δέντρου και είναι ο γονέας**.** Στην heapsort() θέλουμε αυτό το μέγιστο στοιχείο να βρίσκεται στο τελευταίο στοιχείο του ταξινομημένου πίνακα μας. Έτσι, το αλλάξαμε με το τελευταίο στοιχείο και μετά απορρίψαμε αυτό το στοιχείο από το σωρό μειώνοντας το μέγεθος του σωρού. Έτσι, έχουμε τοποθετήσει σωστά το μεγαλύτερο στοιχείο στη σωστή θέση, αλλά με αυτόν τον τρόπο, έχουμε χαλάσει τη ρίζα του σωρού. Τέλος καλούμε την heapify() στην ρίζα του δέντρου χωρίς η ρίζα αυτή την φορά να είναι το μέγιστο στοιχείο. Και συνεχίζουμε μέχρι κάθε στοιχείο του σωρού να τοποθετηθεί σωστά σε ταξινομημένη σειρά.

```
private static void heapify(double [] array,int current,int size ){

        int largest=current; //parent
        int left=2*current+1; //left child
```

```
        int right=2*current+2; //right child

        //check which of the two childs is greater than the current element
        if(left<size && array[left]>array[largest]){
            largest=left;
        }
        if(right<size && array[right]>array[largest]){
            largest=right;
        }

        //swaps the elements
        if(largest!=current){
            double temp=array[current];
            array[current]=array[largest];
            array[largest]=temp;

            heapify(array,largest,size);
        }

    }

    private void heapSort(double [] array){

        for (int i = (array.length / 2) - 1; i >= 0; i--)
            heapify(array,i,array.length);

        for(int i= array.length-1; i>0; --i){
            double temp=array[0];
            array[0]=array[i];
            array[i]=temp;

            heapify(array,0,i); //heapify root(in our case array[0] element)
with i
        }
    }
```

Χρονική Πολυπλοκότητα: και στις τρεις περιπτώσεις(χειρότερη,μέση,καλύτερη) είναι O(nlogn). Με μέτρηση των millisecond που απαιτείται για να υλοποιηθεί ο αλγόριθμος(με χρήση **nanoTime()**πριν και μετά την υλοποίηση του): **0.8097 milliseconds.**



CountingSort:

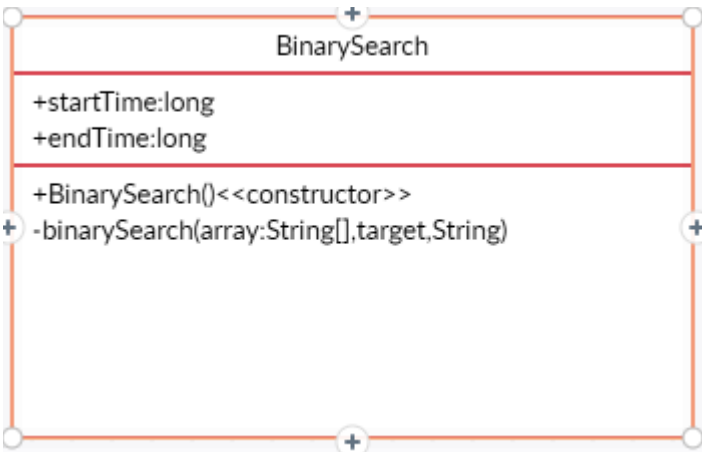
Εδώ πριν ξεκινήσουμε τον αλγόριθμο, στον constructor κάναμε μετατροπή των στοιχείων του πίνακα σε integer καλώντας την μέθοδο **getInt()** η οποία μας έκανε την μετατροπή από double σε integer(στον πλησιέστερο) διότι ο αλγόριθμος υλοποιείται μόνο με integer και αφότου κάναμε την ταξινόμηση μέσω ενός map που φτιάξαμε αντιστοιχήσαμε τον ταξινομημένο integer πίνακα πάλι πίσω σε double πίνακα. Όσον αφορά τον αλγόριθμο, βρήκαμε το max στοιχείο του πίνακα και το χρησιμοποιήσαμε ώστε να φτιάξουμε έναν κενό πίνακα με μέγεθος ίσο με max+1, έπειτα ο κενός πίνακας γέμισε με τις εμφανίσεις κάθε στοιχείου από τον αρχικό μας πίνακα στην συνέχεια αποθηκεύσαμε το αθροιστικό άθροισμα των στοιχείων του πίνακα μετρήσεων των εμφανίσεων το οποίο θα μας βοηθήσει στην σωστή τοποθέτηση των στοιχείων στην σωστή θέση του ταξινομημένου πίνακα. Τέλος αντιστοιχήσαμε κάθε στοιχείο του αρχικού μας πίνακα στον πίνακα μετρήσεων των εμφανίσεων μειώνοντας τον αριθμό του κατά 1 και από εκεί στο output_array το οποίο έχει μέγεθος όσο το αρχικό μας array.

```
private void countingSort(int[] array){


        //find the maximum value on the given array
        int max=array[0];
        for(int i=1; i<array.length; i++){
            if(max<array[i]){
                max=array[i];
            }
        }

        //create an empty index array of max+1 size
        int[] index_array= new int[max+1];

        //fill index array with the number of occurrences
        for(int i = 0; i < index_array.length; i++){
            int counter = 0;
            for(int j = 0; j < array.length; j++){
                if(array[j] == i){
```

```
                counter++;
            }
            index_array[i] = counter;
        }
    }

    //sum up the index array values with previous values
    for (int i = 1; i <= max; i++) {
        index_array[i]= index_array[i]+index_array[i-1];
      }

    //create output array to store sorted values
    int[] output_array = new int[array.length];

    //map the value of the given array with index array and put the value in the output array
    for (int i = array.length-1; i>=0; i--) {
        output_array[index_array[array[i]] - 1] = array[i];
        index_array[array[i]]--; //decrement index array value
    }


    for (int i = 0; i <array.length; ++i){
        array[i] = output_array[i];

    }

}


private int getInt(double i){
    return (int)Math.ceil(i);
}
```

Χρονική πολυπλοκότητα: χειρότερης περίπτωσης: $O(n^2)$ μέσης περίπτωσης και καλύτερης: O(nlogn). Με μέτρηση των millisecond που απαιτείται για να υλοποιηθεί ο αλγόριθμος(με χρήση **nanoTime()**πριν και μετά την υλοποίηση του): **7.2234 milliseconds.**

Σύγκριση **CountingSort-HeapSort:** Όπως φαίνεται και από τους χρόνους που μετρήσαμε πιο γρήγορος στην συγκεκριμένη περίπτωση ειναι ο HeapSort. Αναλυτικότερα όμως στην περίπτωση που γνωρίζουμε ότι τα δεδομένα μας είναι από 0-100 πιο αποδοτικός είναι ο **Counting Sort**, όμως αν θέλουμε να προβλέψουμε με ακρίβεια πότε ο αλγόριθμος μας θα έχει τελειώσει τότε πιο αποδοτικός είναι ο **Heap Sort** καθώς η πολυπλοκότητα του είναι πάντα ίδια σε κάθε περίπτωση.



BinarySearch:

Για το τρίτο ερώτημα ζητήθηκε υλοποίηση των αλγορίθμων **Binary Search** και **Interpolation Search** με βάση το πεδίο date. Ο χρήστης δοθέντος ενός date έπρεπε να κάνουμε αναζήτηση στο text και αν υπήρχε το date που έδινε ο χρήστης να επέστρεφε το αντίστοιχο Volume. Εδώ δεν χρησιμοποιήσαμε maps αλλά δουλέψαμε με Vectors. Η **Binary Search** έχει έναν constructor ο οποίος φτιάχνει αντικείμενο **FiletoArray** ώστε να πάρει τα δεδόμενα που χρειαζόμασταν δηλαδή το text, το πεδίο **Volume** και μέσω της **getDate()** τις ημερομηνίες. Έπειτα καλούσε την Binary Search έκανε την αναζήτηση και αν το date υπήρχε επέστρεφε μέσω της **getVolumeVector()** το αντίστοιχο Volume. Όσον Αφορά την **Interpolation Search** ισχύουν τα ίδια για τον constructor όπως στην Binary Search μόνο που χρησιμοποιούσε και την **String Handler()** καθώς ο αλγόριθμος αυτός λειτουργεί μόνο με array αριθμών(long επειδη χρησιμοποιήσαμε unix timestamp) και έτσι έπρεπε να πάρουμε ξεχωριστά κάθε πεδίο του **date** για να κάνουμε την αναζήτηση. Πιο συγκεκριμένα ο constructor της **BinarySearch:**

```
BinarySearch(String filename,String target){
        try{
            FiletoArray ar = new FiletoArray(filename, "Volume");
            String[] array= ar.getDate();


            //position of the given array
            startTime = System.nanoTime();
            int position = binarySearch(array, target);
            endTime = System.nanoTime();
            //match the position of the date to volume

            if(position !=-1){
                System.out.println("Volume for the given date is: " +
ar.getVolumeVector().get(position));
            }
            else
                System.out.println("Date not found");
```

```
        }
        catch(IOException e){}

    }
```

Οι μεταβλητές **endTime** και **startTime** μετράνε τον χρόνο εκτέλεσης του αλγορίθμου σε nanoseconds.

Υλοποίηση αλγορίθμου(μέθοδος binarySearch):Κάνει αναζήτηση σε έναν ταξινομημένο πίνακα διαιρώντας επανειλημμένα το διάστημα αναζήτησης στο μισό. Ξεκινάμε με ένα διάστημα που καλύπτει ολόκληρο τον πίνακα. Εάν η τιμή του κλειδιού αναζήτησης είναι μικρότερη από το στοιχείο στο μέσο του διαστήματος, περιορίζει το διάστημα στο κάτω μισό. Διαφορετικά, περιορίζει το διάστημα στο πάνω μισό. Τέλος ελέγχει επανειλημμένα μέχρι να βρεθεί η τιμή αλλιώς επιστρέφουμε -1 δηλαδή ότι δεν βρέθηκε.Αναλυτικότερα:

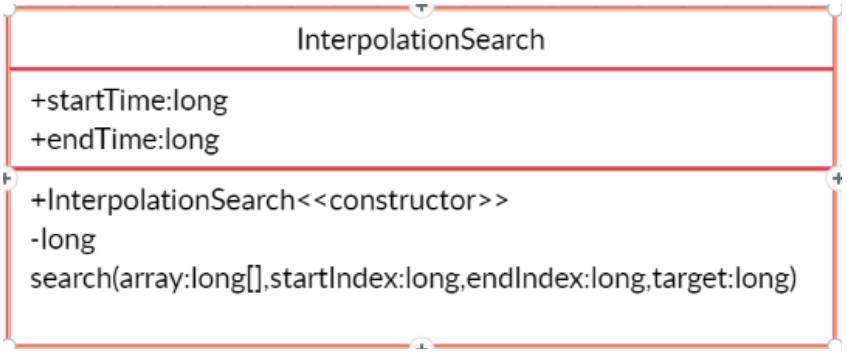
```
private int binarySearch(String[] array,String target){

        int left=0;
        int right=array.length;

        while(left<right){
            int mid=(left+right) /2;

            int compare=target.compareTo(array[mid]);

            if(compare==0){
                return mid;
            }
            else if(compare<0){
                right=mid-1;
            }
            else{
                left=mid+1;
            }
        }

        return -1;

    }
```

Χρονική Πολυπλοκότητα: χειρότερης περίπτωσης και μέσης: O(logn) καλύτερης περίπτωσης: O(1). Με μέτρηση των millisecond που απαιτείται για να

υλοποιηθεί ο αλγόριθμος(με χρήση **nanoTime()**πριν και μετά την υλοποίηση του): **0.0126 milliseconds.**

Υλοποίηση main οπου ο χρήστης δίνει τιμή: Όπως έχει αναφερθεί και παραπάνω η κλάση **StringHandler** διαθέτει μέθοδο η οποία κάνοντας χρήση αμυντικού προγραμματισμού “υποχρεώνει” τον χρήστη να δώσει ημερομηνία στην σωστή μορφή.Γι αυτό το λόγο η **main** διαμορφώνεται ώς εξής:

```
public static void main(String[] args) {

        StringHandler handler = new StringHandler();

        BinarySearch a = new BinarySearch("agn.us.txt",handler.getDate());

        System.out.println("Elapsed time: " + (a.endTime - a.startTime) +"
nanoseconds");

}
```



InterpolationSearch:

Η **Interpolation Search** προϋποθέτει ότι οι τιμές σε έναν ταξινομημένο πίνακα κατανέμονται ομοιόμορφα. Η αναζήτηση παρεμβολής μπορεί να μεταβεί σε διαφορετικές τοποθεσίες ανάλογα με την τιμή του κλειδιού που αναζητείται. Έπειτα κάνει συγκρίσεις με το array[position] και το target και αναλόγως επιστρέφει την θέση αλλιώς καλεί τον εαυτό της αναδρομικά στο κατάλληλο μέρος του υπό πίνακα έως ότου βρεθεί η τιμή ή επιστρέφει -1 δηλαδή ότι δεν βρέθηκε.

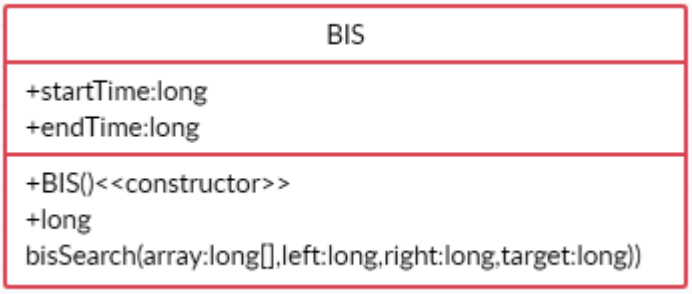
```
private long search(long[] array,long startIndex, long endIndex, long
```

```
target){

        //startIndex will be array[0] for the first time whereas endIndex
will be array[length -1]

        if(array[(int)startIndex] == array[(int)endIndex]){
            return endIndex;
        }
        else if(startIndex <= endIndex && target >= array[(int)startIndex]
&& target <= array[(int)endIndex]){

            //position variable according to the interpolation algorithm
            long position = startIndex + (((endIndex - startIndex) /
(array[(int)endIndex] - array[(int)startIndex]))*(target
-array[(int)startIndex]));


            if(array[(int)position] == target){
                return position; //element found
            }
            /*else if(array[(int)position] == endIndex)
                return search(array,position,endIndex,target);*/
            else if(array[(int)position] < target){
                return search(array,position+1,endIndex,target); //go to the
next element in position of the left part of array
            }
            else if(array[(int)position] > target){
                return search(array,startIndex,position-1,target); //go to
the next element in position of the right part of array(starting from high)
            }


        }

        return -1; //element not found
    }
```

Χρονική Πολυπλοκότητα: χειρότερης περίπτωσης:O(n) καλύτερης περίπτωσης: O(1) και χειρότερης περίπτωσης: O(log log n) . Με μέτρηση των millisecond που απαιτείται για να υλοποιηθεί ο αλγόριθμος(με χρήση **nanoTime()**πριν και μετά την υλοποίηση του): **0.9633 milliseconds.**

Σύγκριση: Η αναζήτηση παρεμβολής λειτουργεί καλύτερα από τη Δυαδική Αναζήτηση για έναν ταξινομημένο και ομοιόμορφα κατανεμημένο πίνακα. Κατά μέσο όρο, η αναζήτηση παρεμβολής πραγματοποιεί συγκρίσεις log (log (n)) (εάν τα στοιχεία κατανέμονται ομοιόμορφα), όπου n είναι ο αριθμός των στοιχείων που πρέπει να αναζητηθούν. Όσο αναφορά το Data Set, όσο μεγαλύτερο το data Set τόσο πιο γρήγορος είναι ο interpolation search.

Η μέθοδος main διαμορφώνεται παρόμοια με της **binary search.**



Binary Interpolation Search:

Η **Binary Interpolation Search** είχε έναν constructor ο οποίος φτιάχνει αντικείμενο **FiletoArray** ώστε να πάρει τα δεδομένα που χρειαζόμασταν δηλαδή το text, το πεδίο Volume και μέσω της **getDate()** τις ημερομηνίες. Στην συνέχεια χρησιμοποιεί την **StringHandler**() καθώς ο αλγόριθμος αυτός λειτουργεί μόνο με integer array και έτσι έπρεπε να πάρουμε ξεχωριστά κάθε πεδίο του date για να κάνουμε την αναζήτηση. Τέλος καλούσε την **bisSearch** έκανε την αναζήτηση και αν το date υπάρχει επιστρέφει μέσω της **getVolumeVector**() το αντίστοιχο Volume. Πιο συγκεκριμένα όσον αφορά τον αλγόριθμο ακολουθήθηκε ακριβώς ο αλγόριθμος του ηλεκτρονικού βιβλίου χωρίς καμία αλλαγή.

```
private long bisSearch(long[] array, long left, long right, long target){


    if(left>right || (left==right && array[(int)left]!=target)){
        return -1;
    }
    else if(left==right && array[(int)left] == target){
        return left;
    }

    long position = (target-array[(int)left])/(array[(int)right] -array[(int)left]);

    long mid = left + position*(right-1);

    int i =1;

    if(target > array[(int)mid]){
        long next = mid + i*(int)Math.sqrt(array.length);

        while(true){
```

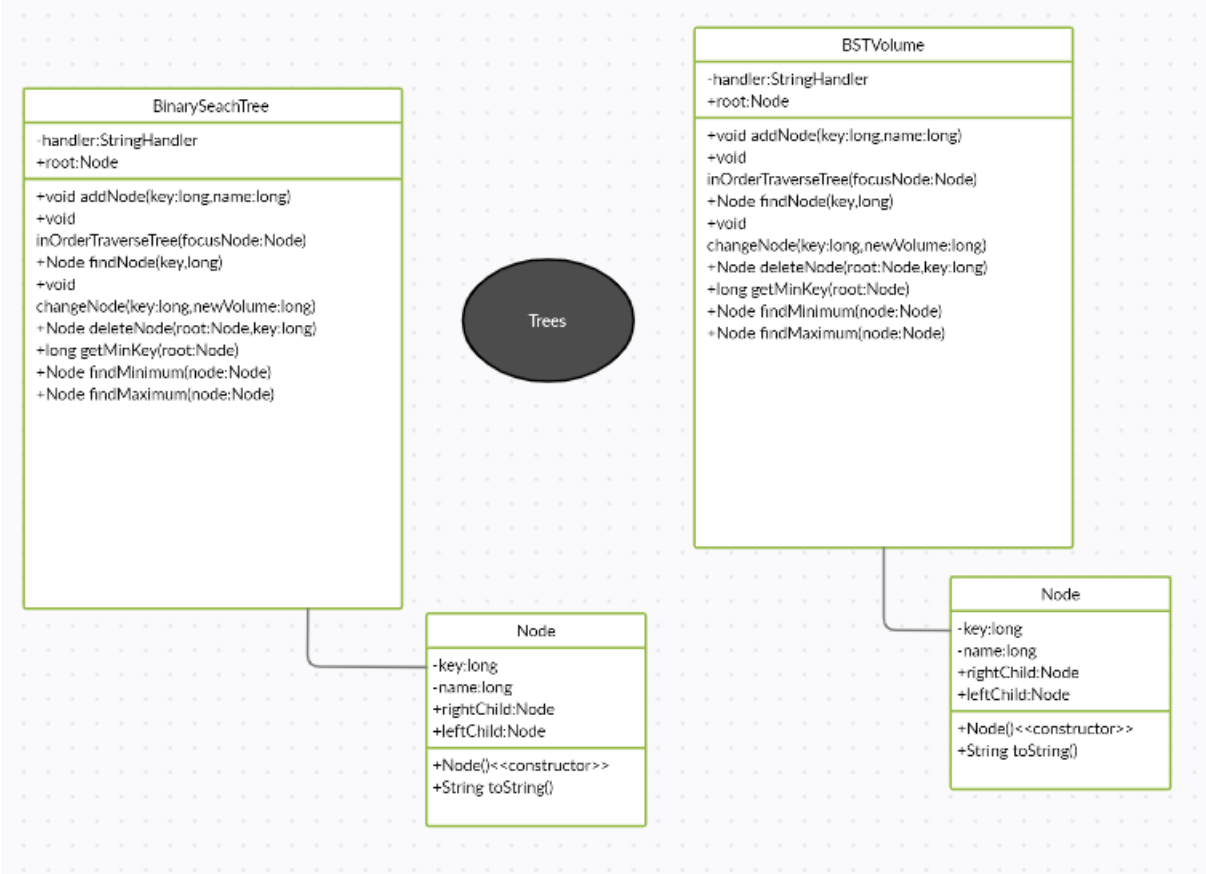
```
            if(next>right || target<array[(int)next])
                break;

            if(target == array[(int)next]){
                return next;
            }

            i = i++;

        }
        left = mid + (i-1)*(int)Math.sqrt(array.length) + 1;

        right = Math.min(right,next -1);
        return bisSearch(array,left,right,target);
    }
    else if(target<array[(int)mid]){
        long next = mid - i*(int)Math.sqrt(array.length);

        while(true) {


            if(next <left || target>array[(int)next])
                break;

            if(target == array[(int)next]){
                return next;
            }

            i = i++;
        }
        right = mid -(i-1)*(int)Math.sqrt(array.length) - 1;
        left = Math.max(left,next + 1);

        return bisSearch(array,left,right,target);
    }
    else{

        return mid;
    }

}
```

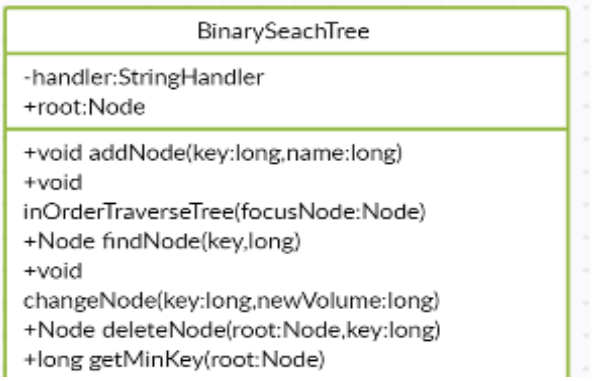
Χρονική Πολυπλοκότητα: μέση περίπτωση: O(log log n), χειρότερη: O($\sqrt{n}$)

Χρονική Πολυπλοκότητα: βελτίωση της χειρότερης περίπτωσης σε O(logn) με αλλαγή του μετρητή από i = i+1 σε i= 2×i

Σύγκριση ως προς τους χρόνους χειρότερης περίπτωσης: με την αλλαγή του μετρητή i αντί να αυξάνεται γραμμικά αυξάνεται εκθετικά με αποτέλεσμα να κάνει ολοένα και μεγαλύτερα άλματα κρατώντας το αριστερό άκρο σταθερό συνεπώς το τελευταίο υποδιάστημα να είναι πολύ πιο μεγάλο από $\sqrt{n}$. Μόλις βρεθεί αυτό το υποδιάστημα εφαρμόζουμε δυική αναζήτηση στα στοιχεία μέσα σε αυτό που απέχουν κατά $\sqrt{n}$ και έτσι προκύπτει το ζητούμενο υποδιάστημα μεγέθους $\sqrt{n}$.

Δευτερο Μερος (Δεντρα + hash)





Πρώτο Ερώτημα:

Όσον αφορά το πρώτο ερώτημα του δεύτερου μέρους ζητήθηκε να υλοποιηθεί ένα Δυαδικό Δέντρο Αναζήτησης το οποίο θα περιείχε σαν εγγραφή τα πεδία **Date,Volume** και το **ΔΔΑ** θα διατάσσεται με βάση το πεδίο **Date**. Πιο συγκεκριμένα για το **(1) ερώτημα** όσον αφορά την απεικόνιση του δέντρου με ενδο-διατεταγμένη διάσχιση στην μέθοδο **addNode()** η προσθήκη κάθε καινούργιας εγγραφής **Date-Volume** έγινε με σύγκριση του κλειδιού(key) το οποίο στην περίπτωση μας ήταν το Date, και η ενδο-διατεταγμένη διάσχιση υλοποιήθηκε από την μέθοδο **inOrderTraverseTree()**.

```
public void inOrderTraverseTree(Node focusNode){
        if(focusNode != null){

            inOrderTraverseTree(focusNode.leftChild);
            System.out.println(focusNode);

            inOrderTraverseTree(focusNode.rightChild);

        }
}
```

Στο **(2) ερώτημα** ζητήθηκε η αναζήτηση όγκου με βάση μιας δοθείσα ημερομηνία από τον χρήστη η οποία υλοποιείται με την μέθοδο **findNode().** Η **findNode()** δέχεται ώς όρισμα ένα κλειδί **key(date για εμας)** και με **while loop** έκανε συγκρίσεις με βάση την δοθείσα ημερομηνία και πήγαινε στο κατάλληλο μέρος του δέντρου για να ψάξει.

```
public Node findNode(long key){
        Node focusNode = root;

        while(focusNode.key != key){
            if(key < focusNode.key){
                focusNode = focusNode.leftChild;
            } else{
                focusNode = focusNode.rightChild;
            }

            if(focusNode == null){
```

```
                return null;
            }
        }
        return focusNode;
}
```

Στο **(3) ερώτημα** ζητήθηκε η τροποποίηση της τιμής **volume** με βάση την ημερομηνία που δινόταν από τον χρήστη αυτό υλοποιήθηκε από την **changeVolume()** η οποία καλεί την **findNode()** για να ελέγξει αν η ημερομηνία υπάρχει και έπειτα αν υπάρχει την αντικαθιστά με την νέα τιμή που θα δοθεί από τον χρήστη.

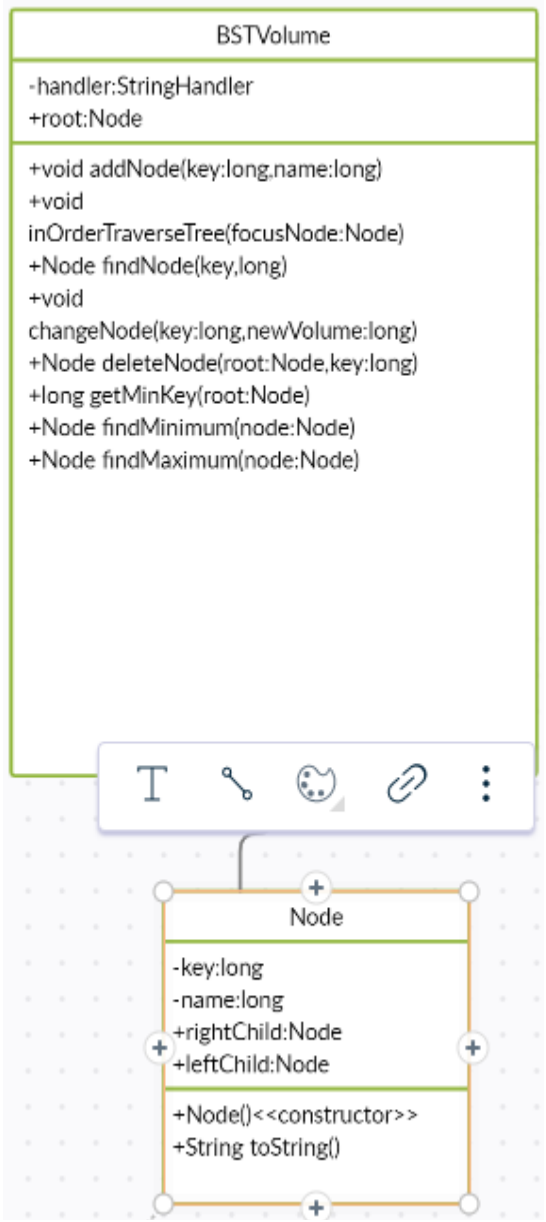
```
public void changeNode(long key, long newVolume){
        Node getFocusNode = findNode(key);

        if(getFocusNode != null){
            getFocusNode.name = newVolume;
            System.out.println(getFocusNode.toString());
        }else{
            System.out.println("Date not found");
        }
}
```

Στο **(4) ερώτημα** ζητείται η διαγραφή μιας εγγραφής με βάση την δοθείσα ημερομηνία από τον χρήστη αυτό υλοποιήθηκε με την **deleteNode()** η οποία ελέγχει με βάση την ημερομηνία ποιον κόμβο πρέπει να ελευθερώσει(διαγράψει) ώστε ο κόμβος ο οποίος έπρεπε να διαγραφεί αντικαθίσταται με τον διάδοχο ή προκάτοχο του στη σειρά εώς ότου η εγγραφή να διαγραφεί και έπειτα ελευθερώνουμε τον εκχωρημένο χώρο.

```
public Node deleteNode(Node root, long key){

        if(root == null){
            return root;
        }
        if(key < root.key){
            root.leftChild = deleteNode(root.leftChild,key);
        }
        else if (key > root.key){
            root.rightChild = deleteNode(root.rightChild, key);
        }
        else{
            if(root.leftChild == null){
                return root.rightChild;
            }
            else if(root.rightChild ==null)
```

```
                return root.leftChild;

            root.key = getMinKey(root.rightChild);
            root.rightChild = deleteNode(root.rightChild,root.key);
        }

        return root;

    }
```

Οι παραπάνω μέθοδοι χρησιμοποιούνται στο Menu που κατασκευάσαμε και θα παρουσιάσουμε αργότερα.



Δεύτερο ερώτημα:

Όσον αφορά το δεύτερο ερώτημα ζητήθηκε το ΔΔΑ να περιέχει ώς εγγραφή τα πεδία **Date,Volume** αλλά αυτή την φορά η διάταξη του δέντρου να γίνει με βάση το πεδίο **Volume**. Αυτή η αλλαγή έγινε στην κλάση Node η οποία έθεσε στο **key** το **name** ώστε να γίνει η δημιουργία του δέντρου με βάση το Volume.

```
class Node {

            long key;
            long name;

            Node leftChild;
            Node rightChild;

            Node(long key, long name){
                this.key = name;
                this.name=key;
            }


            public String toString(){
                  return "Volume: " + key + " Date: " +handler.dateFormat(name);
            }
}
```

Στο (1) ερώτημα ζητήθηκε η εύρεση **ημέρας/ημερών** με βάση **την ελάχιστη τιμή όγκου volume** αυτό υλοποιήθηκε με την μέθοδο **findMinimum()** η οποία έψαχνε μόνο στο αριστερό μέρος του δέντρου καθώς με βάση τον ορισμό δημιουργίας ενός ΔΔΑ η μικρότερη τιμή πάει στα αριστερά του δέντρου και η μεγαλύτερη στα δεξιά.

```
public Node findMinimum(Node node){
            if(node.leftChild != null){
              return findMinimum(node.leftChild);
            }
            return node;
}
```

Στο (2) ερώτημα ζητήθηκε το αντίθετο, δηλαδή **η εύρεση ημέρας/ημερών με βάση την μέγιστη τιμή όγκου volume** αυτό υλοποιήθηκε με την μέθοδο **findMaximum()** η οποία ψάχνει μόνο στο δεξιό μέρος του δέντρου για τον λόγο που αναφέρεται στο πρώτο ερώτημα.

```
public Node findMaximum(Node node){
            if(node.rightChild != null){
              return findMaximum(node.rightChild);
            }
            return node;
}
```

Και στα δύο ερωτήματα (Α) και (Β) στην κλάση **Node()** στην οποία γινόντουσαν οι δηλώσεις των μεταβλητών φτιάξαμε μια μέθοδο **toString()** ώστε να επιστρέψει στον χρήστη ότι ζητήθηκε και εκεί έγινε η τροποποίηση πάλι του **Date** στην κατάλληλη μορφή μέσω της μεθόδου **dateFormat()** της κλάση **StringHandler().**

Hash με αλυσιδες:

Σε αυτό το ερώτημα του δεύτερου μέρους μας ζητούνταν η υλοποίηση Hashing με αλυσίδες. Γενικά η αλυσίδα είναι μια τεχνική που χρησιμοποιείται για την αποφυγή συγκρούσεων σε κατακερματισμούς. Μια σύγκρουση συμβαίνει όταν δύο πλήκτρα κατακερματιστούν στον ίδιο δείκτη σε έναν πίνακα κατακερματισμού. Οι συγκρούσεις είναι ένα πρόβλημα επειδή κάθε υποδοχή σε έναν πίνακα κατακερματισμού υποτίθεται ότι αποθηκεύει ένα μόνο στοιχείο. Τα πλεονεκτήματα είναι:

1) Απλό στην εφαρμογή.

2) Ο πίνακας Hash δεν γεμίζει ποτέ, μπορούμε πάντα να προσθέσουμε περισσότερα στοιχεία στην αλυσίδα.

3) Λιγότερο ευαίσθητο στη συνάρτηση κατακερματισμού ή στους παράγοντες φόρτωσης.

4) Χρησιμοποιείται κυρίως όταν είναι άγνωστο πόσα και πόσο συχνά μπορούν να εισαχθούν ή να διαγραφούν τα πλήκτρα.

Πιο συγκεκριμένα εμείς μέσω της κλάσης **DateVolume** θέταμε και επιστρέφαμε τα **date-volume** που θέλαμε.

Για να αποθηκεύουμε κάθε στοιχείο που θέλαμε στον πίνακα κατακερματισμού χρησιμοποιήσαμε μια συνδεδεμένη λίστα την <**DateVolume**>. Έπειτα μέσω της **HashFunction :**

```
public int HashFunction(String date) {

        int value = 0;
        int counter=0;
```

```
        for(int i=0; i<date.length();i++) {
            value=value+(int)date.charAt(i);
            counter++;
        }
        return value%(counter+1);
    }
```

υλοποιήσαμε την συνάρτηση κατακερματισμού και τέλος με τις μεθόδους **put():**

```
public void put(String date, int volume) {
        int index = HashFunction(date);
        LinkedList<DateVolume> items = hashTable[index];

        if(items == null) {
            items = new LinkedList<DateVolume>();

            DateVolume item = new DateVolume();
            item.date = date;
            item.volume= volume;

            items.add(item);

            hashTable[index] = items;
        }
        else {
            for(DateVolume item : items) {
                if(item.date.equals(date)) {
                    item.volume = volume;
                    return;
                }
            }

            DateVolume item = new DateVolume();
            item.date = date;
            item.volume=volume;

            items.add(item);
        }
}
```

findVolume():

```
public int findVolume(String date) {
        int v = 0;
        int index = HashFunction(date);
        for(DateVolume item:hashTable[index]) {
            if(item.date.equals(date)) {
                v=item.volume;
            }
        }
        return v;
}
```

delete():

```
public void delete(String date) {
        int index = HashFunction(date);
        LinkedList<DateVolume> items = hashTable[index];

        if(items == null)
            return;

        for(DateVolume item : items) {
            if (item.date.equals(date)) {
                items.remove(item);
                return;
            }
        }
}
```

και **changeTemp():**

```
public void changeTemp(String date,int volume) {
        int index = HashFunction(date);
        LinkedList<DateVolume> items = hashTable[index];

        if(items == null)
            return;

        for(DateVolume item : items) {

            if (item.date.equals(date)) {

                item.setVolume(volume);
```

```
                return;
            }
        }
    }
```

υλοποιήσαμε τα ζητούμενα κάθε ερωτήματος που θα έπρεπε να περιέχει το **Menu()**.

Αρχειο Menu

Το αρχείο Menu περιλαμβάνει την κλάση **Menu** και το **interface GoBackToMenu** το οποίο κατασκευάστηκε για να γίνει δυνατή η αλλαγή ορίσματος μεθόδου για την **goBackToMenu().**

Αρχικά η κλάση **Menu** περιλαμβάνει instances απο objects που θα χρειαστούμε όπως (**StringHandler, Scanner, BinarySearchTree...)** και έναν constructor που φορτώνει τις τιμές στα δέντρα και στο hash και καλεί την μέθοδο **treeOrHash()** το οποίο είναι και το αρχικό μενού(ο χρήστης επιλέγει αν θέλει να γίνει αναπαράσταση με δέντρα ή hash).

```
public class Menu {


    StringHandler handler = new StringHandler();
    GoBackToMenu goBack;
    Scanner input = new Scanner(System.in);
    String stringInput;
    BSTVolume volumeTree = new BSTVolume();
    BinarySearchTree tree = new BinarySearchTree();
    int intInput = 0;
    FiletoArray ar;
    Hash hashTable = new Hash();

    Menu(){
        try{
            ar = new FiletoArray("agn.us.txt","Volume");


            long[] array = handler.stringToIntArray(ar.getDate());
            String[] arrayDate = ar.getDate();

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

```
                tree.addNode(array[i], ar.getVolumeVector().get(i));
            }

            long[] array_volume = handler.stringToIntArray(ar.getDate());

            Vector<Integer> getVolume = ar.getVolumeVector();

            for(int i=0; i<getVolume.size(); i++) {
                volumeTree.addNodeVolume(array_volume[i],getVolume.get(i));
            }

            for(int i=0; i<arrayDate.length; i++){
                hashTable.put(arrayDate[i], getVolume.get(i));
            }

            treeOrHash();


        }catch(IOException e){
            e.printStackTrace();
        }

    }
}
```

Μέθοδος **treeOrHash():**

```
private void treeOrHash(){
       Scanner input = new Scanner(System.in);
       while(true){
           System.out.print("\033[H\033[2J");
           System.out.flush();
           System.out.println("Choose Tree or Hash(Type \"Tree\" or \"Hash\")");
           System.out.print(">> ");
           stringInput = input.nextLine();

           if(stringInput.equals("Tree")){
               initialMenu();
               break;
           }
           else if(stringInput.equals("Hash")){
               hashMenu();
               break;
           }
       }
       input.close();
}
```

Κάνοντας χρήση **αμυντικού προγραμματισμού** με while loop “αναγκάζουμε” τον χρήστη να δώσει σωστή επιλογή(ή **Tree** ή **Hash**). Έπειτα άν δώσει την επιλογή Tree καλούμε την μέθοδο **initialMenu()** ενώ αν δώσει Hash καλούμε την μέθοδο **hashMenu()**.

Μέθοδος **initialMenu():**

```
private void initialMenu(){
        while(true){
            System.out.print("\033[H\033[2J");
            System.out.flush();

            System.out.println("Choose tree representation(Type integer between 1 and 4)...");
            System.out.println("1 --> BST which keeps the record Date,Volume sorted by Date: ");
            System.out.println("2 --> BST which keeps the record Date,Volume sorted by Volume: ");
            System.out.println("3 --> Go back <-|");
            System.out.println("4 --> exit");
            System.out.print(">> ");

            stringInput = input.nextLine();

            try {
                intInput = Integer.parseInt(stringInput);

                if(intInput <= 0 || intInput > 4){
                    System.out.print("\033[H\033[2J");
                    System.out.flush();
                    System.out.println("Integer should be between 1 and 4");
                }
                else
                    break;

            }catch (Exception e){
                System.out.print("\033[H\033[2J");
                System.out.flush();
                System.out.println("Please type an integer from 1-4");
            }
        }

        initialChoiceHandler();
}
```

Πάλι με την χρήση αμυντικού προγραμματισμού (και με την χρήση **try catch**) “υποχρεώνουμε” τον χρήστη να δώσει τιμή ακεραίου από το 1-4 όπου καλείται η μέθοδος **initialChoiceHandler()**:

```
private void initialChoiceHandler(){
        switch (intInput) {

            case 1:
                createMenu();
                break;
            case 2:
                createVolumeMenu();
                break;
            case 3:
                System.out.print("\033[H\033[2J");
                System.out.flush();
                treeOrHash();
                break;
            case 4:
                System.exit(0);
                break;
        }
    }
```

- Επιλογή 1: Ο χρήστης έχει ζητήσει την αναπαράσταση ΔΔΑ με βάση το πεδίο ημερομηνία(**Date**) και καλείται η μέθοδος **createMenu():**
    - Μέθοδος **createMenu()** και **choiceHandler()** που καλείται από αυτή**:**

```
private void createMenu(){

        while(true){
            System.out.print("\033[H\033[2J");
            System.out.flush();

            System.out.println("Main menu with BST: (pick from 1 to 6)");
            System.out.println("1 --> Display BST with in order traverse");
            System.out.println("2 --> Search for a Volume by date");
            System.out.println("3 --> Change Volume of a date");
            System.out.println("4 --> Delete a record by date");
            System.out.println("5 --> Go back to initial Menu <-|");
            System.out.println("6 --> exit");

            System.out.print(">> ");
```

```
            stringInput = input.nextLine();

            try {
                intInput = Integer.parseInt(stringInput);

                if(intInput <= 0 || intInput > 6){
                    System.out.print("\033[H\033[2J");
                    System.out.flush();
                    System.out.println("Integer should be between 1 and 6");
                }
                else
                    break;

            }catch (Exception e){
                System.out.print("\033[H\033[2J");
                System.out.flush();
                System.out.println("Please type an integer from 1-6");
            }

        }
        choiceHandler();
    }

    //implement choices of bst menu
    private void choiceHandler(){

        goBack = ()->createMenu();
        String date;
        long dateToInt;
        switch (intInput) {

            case 1:
                System.out.println("You picked display BST");
                tree.inOrderTraverseTree(tree.root);
                goBackToMenu();

                break;
            case 2:
                dateToInt = handler.stringToUnixTime(handler.getDate());

                System.out.println(tree.findNode(dateToInt));

                goBackToMenu();

                break;
            case 3:
                date = handler.getDate();
```

```
                dateToInt = handler.stringToUnixTime(date);

                getInteger();

                tree.changeNode(dateToInt,intInput);

                goBackToMenu();

                break;
            case 4:
                date = handler.getDate();
                dateToInt = handler.stringToUnixTime(date);

                tree.deleteNode(tree.root,dateToInt);

                System.out.println("Volume of date: " + date + " deleted.");
                goBackToMenu();
                break;

            case 5:
                System.out.print("\033[H\033[2J");
                System.out.flush();
                initialMenu();

                break;

            case 6:
                System.exit(0);
                break;
            default:
                break;
        }
    }
```

Ζητάει από τον χρήστη να επιλέξει αν η αναπαράσταση θα γίνει με ενδο-διατεταγμένη διάχηση, αναζήτηση όγκου με βάση ημερομηνίας, αλλαγή εγγραφής με βάση ημερομηνία και διαγραφή εγγραφής με βάση αυτή. Ενώ υπάρχουν και δύο ακόμα επιλογές για επιστροφή στο προηγούμενο Menu και έξοδος απο το πρόγραμμα. Στις προηγούμενες επιλογές αφου γίνει αναπαράσταση του αποτελέσματο καλείται και η μέθοδος **goBackToMenu()** η οποία επιτρέπει να επιστρέψει στο Menu ο χρήστης με την εντολή “back” και “exit” αντίστοιχα. Επειδή η **goBackToMenu()** χρησιμοποιεί την μέθοδο **goback()** του interface θέτουμε με την χρήση lambda(arrow function) την τιμή της μεθόδου ίση με την μέθοδο **createMenu()** για επιστροφή σ αυτο το μενού όταν καλεστεί.

- Επιλογή 2: Ο χρήστης έχει ζητήσει την αναπαράσταση ΔΔΑ με βάση το πεδίο ημερομηνία(**Volume**) και καλείται η μέθοδος **createVolumeMenu():**
    - Μέθοδος **createVolumeMenu():**

```
private void createVolumeMenu(){
        goBack = ()->createVolumeMenu();
        while(true){
            System.out.print("\033[H\033[2J");
            System.out.flush();

            System.out.println("1 --> Find Date sorted by the minimum Volume: ");
            System.out.println("2 --> Find Date sorted by the maximum Volume: ");
            System.out.println("3 --> Go back to initial Menu <-|");
            System.out.println("4 --> exit");
            System.out.print(">> ");


            stringInput = input.nextLine();

            try {
                intInput = Integer.parseInt(stringInput);

                if(intInput <= 0 || intInput > 3){
                    System.out.print("\033[H\033[2J");
                    System.out.flush();
                    System.out.println("Integer should be between 1 and 3");
                }
                else
                    break;

            }catch (Exception e){
                System.out.print("\033[H\033[2J");
                System.out.flush();
                System.out.println("Please type an integer from 1-3");
            }

        }

        switch (intInput){
            case 1:
               System.out.println(volumeTree.findMinimum(volumeTree.root));
               goBackToMenu();
               break;

            case 2:
               System.out.println(volumeTree.findMaximum(volumeTree.root));
               goBackToMenu();
               break;
```

```
            case 3:
                System.out.print("\033[H\033[2J");
                System.out.flush();
                initialMenu();
                break;
            case 4:
                System.exit(0);
                break;
        }

    }
```

Διατηρόντας την ίδια μεθοδολογία με την **createMenu()** ο χρήστης πρέπει να διαλέξει αν θα γίνει Εύρεση ημέρας/ημερών με την ελάχιστη τιμή όγκου συναλλαγών ή με την μέγιστη και αναλόγως θα καλέσει τις μεθόδους του αντικειμένου **volumeTree** (**findMinimum** και **findMaximum** αντίστοιχα).ενώ δίνεται η επιλογή επιστροφής στο προηγούμενο Menu(createMenu) και έξοδος από την εφαρμογή. Δίνεται επιπλέον η δυνατότητα επιστροφής στο ίδιο μενού ή έξοδο μετα την εμφάνιση των αποτελεσμάτων(η μέθοδος **goback()** ισούται με την **createVolumeMenu()** αυτή την φορά).

- Επιλογή 3: Επιστροφή στην επιλογή μενού (Tree ή Hash)
- Επιλογή 4: Έξοδος από την εφαρμογή

Hashing με αλυσίδες: Αν ο χρήστης πληκτρολογήσει την λέξη “Hash” στο **treeOrHash()** τότε θα κληθεί η μέθοδος **hashMenu()** ή οποία θα μεταφέρει στον χρήστη 5 επιλογές τις οποίες διαχειρίζεται με την μέθοδο **hashMenuHandler()**.

```
private void hashMenu(){

        while(true){
            System.out.print("\033[H\033[2J");
            System.out.flush();

            System.out.println("Main hash menu(choose between 1 and 5)...");
            System.out.println("1 --> Search for a Volume based on a date");
            System.out.println("2 --> Change Volume according to a date");
            System.out.println("3 --> Delete Volume based on a date");
            System.out.println("4 --> Go back to \"Tree or Hash\" Menu <-|");
            System.out.println("5 --> exit");
            System.out.print(">> ");
```

```
            stringInput = input.nextLine();

            try {

                intInput = Integer.parseInt(stringInput);

                if(intInput <= 0 || intInput > 5){
                    System.out.print("\033[H\033[2J");
                    System.out.flush();
                    System.out.println("Integer should be between 1 and 5");
                }
                else
                    break;

            }catch (Exception e){
                System.out.print("\033[H\033[2J");
                System.out.flush();
                System.out.println("Please type an integer from 1-5");
            }

        }

        hashMenuHandler(intInput);

    }
```

- Επιλογή 1: Αναζήτηση όγκου με βάση ημερομηνία που δίνεται απο τον χρήστη. Υλοποιείται στο **case 1** της **hashMenuHandler()**

```
case 1:
                givenString = handler.getDate();
                result = hashTable.findVolume(givenString);

                if(result != 0)
                    System.out.println("Volume of " + givenString + " is: " + result);
                else
                    System.out.println("This date does not exist");

                goBackToMenu();
                break;
```

Αρχικά καλούμε το **handler.getDate()** οπου επιστρέφεται μια ημερομηνία τύπου string (yyyy-mm-dd) αφου ο χρήστης έχει αναγκαστεί να την δώσει σ αυτο το format, η οποία αποθηκεύεται στην μεταβλητή τύπου string **givenString**. Έπειτα καλούμε την μέθοδο **findVolume()** του αντικειμένου **hashTable** (τύπου **hash**) το οποίο

αποθηκεύουμε στην τύπου int μεταβλητή result. Άν το **result** δέν ίσο με το 0 τότε σημαίνει ότι βρέθηκε ο όγκος και επιστρέφεται, αλλιώς τυπώνει οτι η ημερομηνία αυτή δεν υπάρχει ενώ παράλληλα δίνει την επιλογή για επιστροφή στο μενού επιλογών με την μέθοδο **goBackToMenu()** (αφού πρώτα έχουμε θέσει την μέθοδο **goBack()** ίση με **hashMenu()** μέσω **arrow function**).

- Επιλογή 2: Η δεύτερη επιλογή επιτρέπει στο χρήστη να αλλάξει μια τιμή όγκου με βάση ημερομηνία που δίνεται απο τον χρήστη. Για αυτό το λόγο ακολουθόντας αρχικά παρόμοια μεθοδολογία όπως και στην πρώτη επιλογή αναζητούμε (με την μέθοδο **findVolume()**) αρχικά την ημερομηνία και αν αυτή υπάρχει (result διάφορο του 0) τότε καλούμε την μέθοδο **changeTemp()** του αντικειμένου **hashTable** τύπου **hash**.

```
case 2:
                givenString = handler.getDate();
                result = hashTable.findVolume(givenString);

                if(result != 0){
                    getInteger();
                    hashTable.changeTemp(givenString,intInput);
              System.out.println("Volume of date: " + givenString + " changed to " + intInput + ".");
                }
                else{
                    System.out.println("This date does not exist!");
                }
                goBackToMenu();
                break;
```

- Επιλογή 3: Στην τρίτη επιλογή ο χρήστης διαγράφει ένα στοιχείο με βάση ημερομηνία που δίνει ο ίδιος. Όπως και στα προηγούμενα ερωτήματα ελέγχουμε αν αυτή η ημερομηνία υπάρχει (με την μέθοδο **findVolume()**) και αν υπάρχει (result διάφορο του 0) τότε καλούμε την μέθοδο **delete()** του αντικειμένου **hashTable** τύπου **hash**.

```
case 3:
                givenString = handler.getDate();
                result = hashTable.findVolume(givenString);

                if(result != 0){
                    hashTable.delete(givenString);
      System.out.println("Record with date: " + givenString + " and volume: " + result + " deleted.");
                }
```

```
else{
    System.out.println("This date does not exist!");
}
goBackToMenu();
break;
```

- Στις επιλογές 4 και 5 δίνεται η δυνατότητα στο χρήστη να επιστρέψει στο προηγούμενο μενού (**Tree** ή **hash**) και να κάνει έξοδο απο την επιλογή αντίστοιχα.

Σύνοψη: Πιστεύουμε ότι όλες οι μέθοδοι του project που ζητήθηκαν λειτουργούν εκτός απο ενα λάθος στην BIS όπου καποιες τιμές δεν τρέχουν.

Τελος