Τμήμα Μηχανικών Η/Υ & Πληροφορικής

Πανεπιστήμιο Πατρών

Μάθημα Δομών Δεδομένων

Υποχρεωτική Εργασία Εαρινού Εξαμήνου 2024

Συμμετέχοντες:

Καραμαλίκης Ανδρέας, Α.Μ: 1100562, Έτος: 2ο

Κατσαρός Γεώργιος, Α.Μ: 1093386, Έτος: 2ο

Κώτσαλος Ιωάννης, Α.Μ: 1100603, Έτος: 2ο

Παπαγιαννούλης Γεώργιος, Α.Μ: 1100660, Έτος: 2ο

Εαρινό Εξάμηνο Ακαδημαϊκού Έτους 2023-2024

Περιεχόμενα:

Εισαγωγή

Στόχος της εργασίας αυτής είναι η εφαρμογή και η σύγκριση διαφόρων αλγορίθμων ταξινόμησης και αναζήτησης σε ένα σύνολο δεδομένων που αποτελείται από τοπικές καταμετρήσεις γεννήσεων και θανάτων. Οι κύριοι στόχοι είναι η εκτέλεση αποδοτικών αλγορίθμων ταξινόμησης και αναζήτησης σε αυτά τα δεδομένα και η ανάλυση της απόδοσης τους. Για το δεύτερο μέρος, γίνεται χρήση δομών, όπως τα Δέντρα Δυαδικής Αναζήτησης (BST) και οι Πίνακες Κατακερματισμού, για την εφαρμογή αποτελεσματικών τεχνικών αποθήκευσης, ανάκτησης και χειρισμού αυτών των δεδομένων.

Δομή Προγράμματος

Για να οργανώσουμε και να υλοποιήσουμε τις λειτουργίες όπως περιγράφεται,επιλέξαμε να χωρίσουμε τον κωδικά μας σε μερικά βασικά source files και τα αντίστοιχα τους header Files. Συγκεκριμένα, τα βασικά αρχεία μας είναι:

init.h

Το header file *init.h* αυτο ορίζει διάφορες δομές και σταθερές και θέτει της βάσεις για την κατασκευή διαφόρων τμημάτων του προγράμματος. Πιο συγκεκριμένα, ορίζει:

- **struct Row**: Η δομή Row έχει σχεδιαστεί για να αναπαριστά μια μεμονωμένη εγγραφή από το αρχείο δεδομένων εισόδου. Αυτή η δομή παρέχει έναν τρόπο αποθήκευσης και χειρισμού μεμονωμένων εγγραφών πριν από την επεξεργασία τους σε πιο σύνθετες δομές δεδομένων όπως η Region. Η περίοδος αποθηκεύεται σε μορφή ακεραίου *int period* και το όνομα της περιοχής ως ακολουθία χαρακτήρων *string region.* Το αν η εγγραφή αφορά γεννήσεις ή θανάτους αποθηκεύεται ως μεταβλητή boolean, *bool BirthDearh*. Αν η εγγραφή αφορά γεννήσεις, τοτε αποθηκεύεται η τιμη *true* αλλιώς η τιμή *false.* Ο αριθμός των αντίστοιχων γεννήσεων ή θανάτων αποθηκεύεται ως ακέραιος *int cout.*
- **struct Period**: Περιέχει πληροφορίες σχετικά με τις γεννήσεις και τους θανάτους για ένα συγκεκριμένο έτος, *int births, int deaths*.
- **struct Region**: Η δομή Region ενσωματώνει όλα τα σχετικά δεδομένα και τις λειτουργίες που σχετίζονται με μια περιοχή. Αποθηκεύει το όνομα της περιοχής *string name*, έναν πίνακα από δομές Period, *Period periodArray*,* για κάθε έτος και τις συνολικές μετρήσεις των γεννήσεων και των θανάτων, *int totalBirths, int totalDeaths*. Η συνάρτηση *Period *period(int)* παρέχει πρόσβαση στα δεδομένα ενός συγκεκριμένου έτους μετατρέποντας το έτος σε δείκτη του πίνακα *periodArray**. Οι συναρτήσεις *void calculateTotalBirths()* και *void calculateTotalDeaths()* αθροίζουν το συνολικό αριθμό γεννήσεων και θανάτων το οποιο αποθηκεύται στα *int totalBirths* και *int totalDeaths* αντίστοιχα.
- **struct Node**: Η δομή Node έχει σχεδιαστεί για να αποτελεί δομικό στοιχείο για την κατασκευή ενός δυαδικού δέντρου αναζήτησης (BST). Κάθε κόμβος στο δέντρο αναπαριστά μια περιοχή *Region region* και περιέχει αναφορές στα παιδιά του, *list<Node>::iterator left, list<Node>::iterator right* και στον πατρικό του κόμβο *list<Node>::iterator parent*. *bool nullNode*: είναι μια σημαία που δείχνει αν ο κόμβος είναι ένας μηδενικός (*null*) κόμβος. Αυτό χρησιμοποιείται για το χειρισμό ακραίων

περιπτώσεων και για την απλοποίηση ορισμένων λειτουργιών στο δέντρο. Η δομή Node χρησιμοποιείται μονο στο PartII.

basicFunctions.cpp

Το αρχείο *basicFunctions.cpp* περιέχει ορισμούς βασικών συναρτήσεων.

- **void readFile(ifstream&, Row*)**: Διαβάζει δεδομένα από ένα αρχείο CSV και γεμίζει έναν πίνακα Row structs.
- **void printRegionArrayTotalBirths(Region*)**: εκτυπώνει τον συνολικό αριθμό γεννήσεων για κάθε περιοχή.
- **void printRegionArrayTotalDeaths(Region*)**: εκτυπώνει το συνολικό αριθμό θανάτων για κάθε περιοχή.
- **void printDataAtIndex(int, Row*)**: Εκτυπώνει δεδομένα σε ένα συγκεκριμένο δείκτη στον πίνακα των δομών Row.
- **void makeRegionArray(Region*, Row*)**: Αρχικοποιεί έναν πίνακα δομών Region από έναν πίνακα δομών Row.
- **void calcTotalBirths(Region*)**: Υπολογίζει το σύνολο των γεννήσεων για κάθε περιοχή.
- **void calcTotalDeaths(Region*)**: Υπολογίζει το σύνολο των θανάτων για κάθε περιοχή.
- **void makeNodeVector(list<Node>&, Region*)**: Δημιουργεί μια λίστα δομών Node από έναν πίνακα δομών Region.
- **void findRelationByRegion(list<Node>&, list<Node>::iterator, list<Node>::iterator)**: Δημιουργεί σχέσεις γονέα-παιδιού στο BST με βάση τα ονόματα περιοχών.
- **void findRelationByBirths(list<Node>&, list<Node>::iterator, list<Node>::iterator)**: Δημιουργεί σχέσεις γονέα-παιδιού στο BST με βάση το σύνολο των γεννήσεων.
- **void makeTree(list<Node> &, string)**: Κατασκευάζει το BST χρησιμοποιώντας την κατάλληλη συνάρτηση σύγκρισης. Εαν το string της παραμέτρου ειναι *“region”* τοτε η κατασκευή γίνεται βάση των ονομάτων των περιοχών, αν ειναι “*births*” τοτε γίνεται βάση του συνόλου των γεννήσεων.
- **void inorderTraversal(list<Node> &, list<Node>::iterator)**: εκτυπώνει τα ονόματα των περιοχών και το σύνολο των γεννήσεων με ενδο-διατεταγμένη διάσχιση.
- **list<Node>::iterator findOrderSuccessor(list<Node> &, list<Node>::iterator)**: Βρίσκει τον μικρότερο διάδοχο ενός κόμβου. Άρα αν στην δεύτερη παράμετρο μπει το δεξί παιδί ενός κόμβου τότε αυτό που θα επιστραφεί θα είναι η επόμενος αριθμητικά κόμβος. Αυτό είναι χρήσιμο στις διαγραφές κόμβων ώστε να βρούμε με ποιον πρέπει να αντικατασταθεί.
- **int hashingFunction(string, int)**: Υπολογίζει την τιμή κατακερματισμού για ένα δεδομένο κλειδί *string key* και έναν αριθμό κάδων *int m*.
- **int makeHashTable(list<Region>*, Region*):** Δημιουργεί έναν πίνακα κατακερματισμού από έναν πίνακα από δομές Region.

searchFunctions.cpp

Το αρχείο *searchFunctions.cpp* περιλαμβάνει διάφορες συναρτήσεις αναζήτησης που χρησιμοποιούνται στο project. Πιο συγκεκριμένα:

- **int binarySearch(Region*, int, int, int)**: Αυτή η συνάρτηση εκτελεί δυαδική αναζήτηση σε έναν πίνακα από δομές Region, αναζητώντας συγκεκριμένα το πεδίο totalBirths. Επιστρέφει τον δείκτη της περιοχής αν βρεθεί, ή -1 αν δεν βρεθεί.
- **void birthsInRangeBinary(int b1, int b2, Region*)**:Αυτή η συνάρτηση χρησιμοποιεί τη binarySearch για να βρει και να εκτυπώσει περιοχές με συνολικές γεννήσεις εντός του εύρους [b1, b2].
- **int binaryInterpolationSearch(Region*, int, int)**: Αυτή η συνάρτηση εκτελεί αναζήτηση παρεμβολής σε έναν πίνακα από δομές Region, με βάση το πεδίο totalBirths. Επιστρέφει τον δείκτη αν βρεθεί, ή -1 αν δεν βρεθεί
- **void birthsInRangeInterpolation(int b1, int b2, Region*)**:Αυτή η συνάρτηση χρησιμοποιεί την αναζήτηση δυαδικής παρεμβολής για να βρει και να εκτυπώσει περιοχές με συνολικές γεννήσεις εντός του εύρους [b1, b2].
- **int BIS(Region arr[], int x)**: Εκτελεί Δυική Αναζήτηση Παρεμβολής
- **void birthsInRangeBIS(int b1, int b2, Region arr[])**: Αναζητά περιοχές με συνολικό αριθμό γεννήσεων εντός του εύρους [b1, b2] και εκτυπώνει τα αποτελέσματα. Χρησιμοποιεί την συνάρτηση *BIS* που εκτέλει Δυική Αναζήτηση Παρεμβολής.
- i**nt optBIS(Region*, int)**: Αυτή η συνάρτηση αποτελεί μια βελτιστοποιημένη έκδοση της αναζήτησης δυικής παρεμβολής.
- **void birthsInRangeOptBIS(int b1, int b2, Region arr[])**: Η συνάρτηση αυτή χρησιμοποιεί τη βελτιστοποιημένη αναζήτηση δυαδικής παρεμβολής για την εύρεση και την εκτύπωση περιοχών με συνολικές γεννήσεις εντός του εύρους [b1, b2].
- l**ist<Node>::iterator searchTree(list<Node>&, string, list<Node>::iterator)**: Αυτή η συνάρτηση αναζητά μια περιοχή σε ένα δυαδικό δέντρο αναζήτησης (BST) και επιστρέφει τον δείκτη του κόμβου αν βρεθεί. Αν δεν βρεθεί επιστρέφει την θέση μνήμης list.end(). Πραγματοποιεί αναζήτηση συγκρίνοντας τα ονόματα των περιοχών.

sortingFunctions.cpp

Το αρχείο *sortingFunctions.cpp* περιέχει τους ορισμούς των συναρτήσεων ταξινόμησης που χρησιμοποιούμε. Αυτές οι συναρτήσεις χρησιμοποιούνται για την ταξινόμηση ενός πίνακα από δομές Region με βάση διαφορετικά κριτήρια.

QuickSort:

- **int splitArr(Region arr[], int low, int high)**: Χωρίζει τον πίνακα γύρω από ένα στοιχείο.
- **void quickSort(Region arr[], int low, int high)**: Ταξινομεί αναδρομικά τον πίνακα χρησιμοποιώντας τον αλγόριθμο QuickSort.

MergeSort:

- **void merge(Region arr[], int l, int m, int r)**: Συγχωνεύει δύο ταξινομημένους υποπίνακες του arr.

- **void mergeSort(Region arr[], int l, int r)**: Αναδρομική ταξινόμηση του πίνακα χρησιμοποιώντας τον αλγόριθμο MergeSort.

HeapSort:

- **void heapify(Region regionArray[], int n, int i)**: Εκτέλεση Φάσης Δόμησης του πίνακα *regionArray[]* σε σωρό.
- **void heapSort(Region regionArray[], int n)**: Ταξινόμηση του πίνακα χρησιμοποιώντας τον αλγόριθμο HeapSort.

CountingSort:

- **void findMaxDeaths(Region regionArray[])**: εύρεση του μέγιστου αριθμού θανάτων στον πίνακα.
- **void countingSort(Region arrayIn[], Region arrayOut[])**: Ταξινόμηση του πίνακα χρησιμοποιώντας τον αλγόριθμο CountingSort.

PartI.1.cpp

Απλά καλέι της συναρτήσεις **void quickSort(Region arr[], int low, int high)** και **void mergeSort(Region arr[], int l, int r)**, και τυπώνει τα αποτελέσματα και τους χρόνους εκτέλεσης κάθε συνάρτησης.

PartI.2.cpp

Απλά καλέι της συναρτήσεις **void heapSort(Region regionArray[], int n)** και **void countingSort(Region arrayIn[], Region arrayOut[])**, και τυπώνει τα αποτελέσματα και τους χρόνους εκτέλεσης κάθε συνάρτησης.

PartI.3.cpp

Αρχικά ζητάει από τον χρήστη το εύρος τιμών της αναζητησης (*b1, b2*) και καλει τις συναρτήσεις **void birthsInRangeBinary(int b1, int b2, Region*)** και **void birthsInRangeInterpolation(int b1, int b2, Region*)** και τυπώνει τα αποτελέσματα και τους χρόνους εκτέλεσης κάθε συνάρτησης.

PartI.4.cpp

Αρχικά ζητάει από τον χρήστη το εύρος τιμών της αναζητησης (*b1, b2*) και καλει τις συναρτήσεις **void birthsInRangeBIS(int b1, int b2, Region arr[])**: και **void void birthsInRangeOptBIS(int b1, int b2, Region arr[])** και τυπώνει τα αποτελέσματα και τους χρόνους εκτέλεσης κάθε συνάρτησης.

PartII.A.cpp

Περιέχει τις συναρτήσεις που χρειαζόμαστε για την εκτέλεση του Ερωτήματος Α.

- **int PartIIA(Region regionArray[], list<Node> &nodeVector)**: Ζητάει από τον χρήστη τι θελει να να κανει και του εμφανίζει ένα μενού επιλογων. Ανάλογα με την επιλογή του χρήστη εκτελείται και η αντίστοιχη συνάρτηση.
- **void QuestionA1(list<Node>&)**: Καλεί την συνάρτηση *void inorderTraversal(list<Node> &, list<Node>::iterator)* που εκτελεί απεικόνιση του ΔΔΑ με ενδο-διατεταγμένη διάσχιση.
- **void QuestionA2(list<Node> &nodeVector)**: Αναζητά τον αριθμό γεννήσεων για χρονική περίοδο και περιοχή που ζητείται από τον χρήστη. Η αναζήτηση της περιοχης γίνεται με την συνάρτηση *list<Node>::iterator searchTree(list<Node>&, string, list<Node>::iterator).*
- **void QuestionA3(list<Node> &nodeVector)**: Τροποποιεί το πεδίο αριθμού γεννήσεων για χρονική περίοδο και περιοχή που δίνονται από το χρήστη. Καλείται η συνάρτηση *list<Node>::iterator searchTree(list<Node>&, string, list<Node>::iterator)* για να βρεθεί η περιοχή που θελει να επεξεργαστεί ο χρήστης.
- **void QuestionA4(list<Node> &nodeVector)**: Διαγραφή μιας εγγραφής βάσει της περιοχής που δίνεται από το χρήστη. Καλείται η συνάρτηση *list<Node>::iterator searchTree(list<Node>&, string, list<Node>::iterator)* για να βρεθεί η περιοχή που θελει να διαγράψει ο χρήστης. Αν ο κόμβος που θέλουμε να διαγραφεί δεν έχει παιδιά τότε απλά διαγράφεται με την εντολή *nodeVector.erase(found)* και δείχνουμε τον pointer του πατρικού του κόμβου που έδειχνε σε αυτόν στην θέση nodeVector.end() το οποίο το αντιμετωπίζουμε ως θέση χωρίς κόμβο. Αν υπάρχει μόνο αριστερό παιδί τοτε ο κόμβος που θα διαγραφεί πρέπει να αντικατασταθεί με αυτό το αριστερό παιδί του. Αν υπαρχει δεξί παιδί τοτε ο κόμβος που θα διαγραφεί πρεπει να αντικατασταθεί από τον αριστερότερο απόγονο του δεξιού παιδιού του κόμβου που διαγράφουμε τον οποίον τον βρίσκουμε με την συνάρτηση *findOrderSuccessor(nodeVector, found->right)*. Δηλαδή σε κάθε περίπτωση αντικαθηστούμε τον κόμβο που διαγράφουμε με τον αμέσως επόμενο αλφαβητικα κόμβο.

PartII.B.cpp

Περιέχει τις συναρτήσεις που χρειαζόμαστε για την εκτέλεση του Ερωτήματος Β.

- **int PartIIB(list<Node> &nodeVector)**: Ζητάει από τον χρήστη τι θελει να να κανει και του εμφανίζει ένα μενού επιλογων. Ανάλογα με την επιλογή του χρήστη εκτελείται και η αντίστοιχη συνάρτηση.
- **void QuestionB1(list<Node>::iterator, const list<Node>&)**: Εύρεση Περιοχής/Περιοχών με τον ελάχιστο αριθμό γεννήσεων. Αναδρόμικα, βρήσκει τον αριστερότερο κόμβο του δέντρου, αρα και αυτόν με τον ελάχιστο αριθμό γεννησεων.
- **void QuestionB2(list<Node>::iterator, const list<Node>&)**: Εύρεση Περιοχής/Περιοχών με τον μέγιστο αριθμό γεννήσεων. Αναδρόμικα, βρήσκει τον δεξιότερο κόμβο του δέντρου, αρα και αυτόν με τον μέγιστο αριθμό γεννησεων.

PartII.C.cpp

Περιέχει τις συναρτήσεις που χρειαζόμαστε για την εκτέλεση του Ερωτήματος Γ.

- **int PartIIC(list<Region> hashTable[])**: Ζητάει από τον χρήστη τι θελει να να κανει και του εμφανίζει ένα μενού επιλογων. Ανάλογα με την επιλογή του χρήστη εκτελείται και η αντίστοιχη συνάρτηση.
- **void QuestionC1(list<Region> hashTable[])**: Αναζήτηση του αριθμού γεννήσεων για χρονική περίοδο και περιοχή που δίνονται από το χρήστη. Η περίοδος που αναζητείται κατακερματίζεται με την *hashingFunction(searchRegion, M)*. Αυτό μας δίνει και την θέση της περιοχής που ψάχνουμε στον πίνακα κατακερματισμού *hashTable.* Μετά γίνεται σειριακή αναζήτηση στη λιστα αυτής της θέσεις μέχρι να βρεθεί η περιοχή.
- **void QuestionC2(list<Region> hashTable[])**: Τροποποίηση του πεδίου αριθμού γεννήσεων για χρονική περίοδο και περιοχή που δίνονται από το χρήστη. Με τον ίδιο τρόπο με την συνάρτηση *void QuestionC1(list<Region> hashTable[])* γίνεται αναζήτηση της περιοχής που θέλει ο χρήστης να επεξεργαστεί και γίνεται η επεξεργασία.
- **void QuestionC3(list<Region> hashTable[])**: Διαγραφή μιας εγγραφής βάσει της περιοχής που δίνεται από το χρήστη. Με τον ίδιο τρόπο με την συνάρτηση *void void QuestionC1(list<Region> hashTable[])* και *void QuestionC2(list<Region> hashTable[])* γίνεται αναζήτηση της περιοχής που θέλει ο χρήστης να επεξεργαστεί και γίνεται η διαγραφή με την εντολή *hashTable[hash].erase(j)*.

PartII.cpp

Ρωτάει τον χρήστη αν θέλει να χρησιμοποιήσει Δυαδικό Δένδρο Αναζήτησης (bst) ή κατακερματισμό (hash). Αν επιλέξει Δυαδικό Δένδρο Αναζήτησης (bst), τοτε το πρόγραμμα ρωτάει τον χρήστη αν θέλει το δέντρο να είναι βάση των περιοχών (region) ή βαση του συνολικού αριθμού γεννήσεων (births).
Αν επιλέξει region τοτε εκτελειται παράγεται το δέντρο και εκτελείται η συνάρτηση *int PartIIA(Region regionArray[], list<Node> &nodeVector)* μεχρι να επιστραφεί από αυτή τιμή μηδέν, το οποίο δηλώνει ότι ο χρήστης τερμάτισε την λειτουργία της.

Αν επιλέξει births τοτε εκτελειται παράγεται το δέντρο και εκτελείται η συνάρτηση *int PartIIB(list<Node> &nodeVector)* μεχρι να επιστραφεί από αυτή τιμή μηδέν, το οποίο δηλώνει ότι ο χρήστης σταμάτησε την λειτουργία της.
Αν αρχικά επιλέξει hash τότε παράγεται ο πίνακας κατακερματισμού και εκτελείται η συνάρτηση *int PartIIC(list<Region> hashTable[])* μεχρι να επιστραφεί από αυτή τιμή μηδέν, το οποίο δηλώνει ότι ο χρήστης τερμάτισε την λειτουργία της.

Απαντήσεις Ερωτημάτων

PART I: “Sorting and Searching Algorithms”

Για να μετρήσουμε την απόδοση των αλγόριθμων χρησιμοποιούμε την βιβλιοθήκη time.h που περιέχει το clock το οποίο μας επιτρέπει να μετράμε τον αριθμό των κύκλων του επεξεργαστή οι οποίοι πέρασαν κατά την εκτέλεση των συναρτήσεων. Καθώς ο αριθμός αυτός δεν είναι σταθερός από σύστημα σε σύστημα και εξαρτάται από τον φόρτο του επεξεργαστή εκείνης της στιγμής, για να εξαχθεί ένα συμπέρασμα πρέπει να παρθούν πολλές μετρήσεις και να συγκριθεί η μέση τιμή τους.

1. Merge Sort και Quick Sort

Παρακάτω ακολουθούν τα αποτελέσματα που τυπώνονται στην οθόνη μετα την εκτέλεση του προγράμματος

--Non-Sorted Array:-- Northland region: 40191 Auckland region: 391893 Waikato region: 109491 Bay of Plenty region: 70983 Gisborne region: 13041 Hawke's Bay region: 38973 Taranaki region: 27327 Manawatu-Wanganui region: 55497 Wellington region: 111969 Tasman region: 8805 Nelson region: 9861 Marlborough region: 9147 West Coast region: 6747 Canterbury region: 125601 Otago region: 40845 Southland region: 21960 Region not stated or area outside region: 792 New Zealand: 1083162	--QuickSortedArray:- Region not stated or area outside region: 792 West Coast region: 6747 Tasman region: 8805 Marlborough region: 9147 Nelson region: 9861 Gisborne region: 13041 Southland region: 21960 Taranaki region: 27327 Hawke's Bay region: 38973 Northland region: 40191 Otago region: 40845 Manawatu-Wanganui region: 55497 Bay of Plenty region: 70983 Waikato region: 109491 Wellington region: 111969 Canterbury region: 125601 Auckland region: 391893 New Zealand: 1083162	--MergeSortedArray:-- Region not stated or area outside region: 792 West Coast region: 6747 Tasman region: 8805 Marlborough region: 9147 Nelson region: 9861 Gisborne region: 13041 Southland region: 21960 Taranaki region: 27327 Hawke's Bay region: 38973 Northland region: 40191 Otago region: 40845 Manawatu-Wanganui region: 55497 Bay of Plenty region: 70983 Waikato region: 109491 Wellington region: 111969 Canterbury region: 125601 Auckland region: 391893 New Zealand: 1083162

Οι χρόνοι σε κύκλους μετα απο διαδοχικές εκτελέσεις του προγράμματος για κάθε αλγόριθμο τυπώνονται όπως παρακάτω: (2x Intel Pentium CPU G3258 @3.20GHz)

Εκτέλεση 1η	**Εκτέλεση 2η**	**Εκτέλεση 3η**
-----Durations:----- Duration of quickSort: 6 clock ticks Duration of mergeSort: 11 clock ticks	-----Durations:----- Duration of quickSort: 8 clock ticks Duration of mergeSort: 15 clock ticks	-----Durations:----- Duration of quickSort: 7 clock ticks Duration of mergeSort: 17 clock ticks

Παρόλο που οι τιμές δεν είναι σταθερές παρατηρούμε πως ο αλγόριθμος quicksort είναι σχεδόν 2 φορές **γρηγορότερος** του merge sort σε κάθε εκτέλεση του προγράμματος.

2. Heap Sort και Counting Sort

Παρακάτω ακολουθούν τα αποτελέσματα που τυπώνονται στην οθόνη μετα την εκτέλεση του προγράμματος

```
--HeapSortedArray by Deaths:--
Region not stated or area outside region: 939
West Coast region: 5070
Tasman region: 6501
Gisborne region: 7140
Marlborough region: 7218
Nelson region: 7560
Southland region: 14745
Taranaki region: 17424
Hawke's Bay region: 24861
Northland region: 26223
Otago region: 28950
Manawatu-Wanganui region: 36753
Bay of Plenty region: 43290
Waikato region: 55626
Wellington region: 57162
Canterbury region: 78264
Auckland region: 143199
New Zealand: 560928
```

```
--countingSorted Array by Deaths:--
Region not stated or area outside region: 939
West Coast region: 5070
Tasman region: 6501
Gisborne region: 7140
Marlborough region: 7218
Nelson region: 7560
Southland region: 14745
Taranaki region: 17424
Hawke's Bay region: 24861
Northland region: 26223
Otago region: 28950
Manawatu-Wanganui region: 36753
Bay of Plenty region: 43290
Waikato region: 55626
Wellington region: 57162
Canterbury region: 78264
Auckland region: 143199
New Zealand: 560928
```

Οι χρόνοι σε κύκλους μετα απο διαδοχικές εκτελέσεις του προγράμματος για κάθε αλγόριθμο τυπώνονται όπως παρακάτω: (2x Intel Pentium CPU G3258 @3.20GHz)

Εκτέλεση 1η	**Εκτέλεση 2η**	**Εκτέλεση 3η**

`--Durations:--`
`Duration of HeapSort: 7 clock ticks`
`Duration of CountingSort: 3746 clock ticks`
	`--Durations:--`
	`Duration of HeapSort: 9 clock ticks`
	`Duration of CountingSort: 4484 clock ticks`
		`--Durations:--`
		`Duration of HeapSort: 9 clock ticks`
		`Duration of CountingSort: 4411 clock ticks`

Εδώ, παρατηρούμε πως ο αλγόριθμος Heap Sort είναι **γρηγορότερος** του counting sort με μεγάλη διαφορά.

3. Δυαδική Αναζήτηση και Αναζήτηση με Παρεμβολή

Δυαδική Αναζήτηση

Απόδοση μέσης περίπτωσης:

Η δυαδική αναζήτηση έχει μέση χρονική πολυπλοκότητα περίπτωσης O(logn), όπου n είναι ο αριθμός των στοιχείων του ταξινομημένου πίνακα.

Επίδραση της κατανομής των δεδομένων:

Η απόδοση της δυαδικής αναζήτησης δεν επηρεάζεται σημαντικά από την κατανομή του συνόλου δεδομένων, επειδή βασίζεται αποκλειστικά στην ταξινομημένη σειρά των στοιχείων και εκτελεί μια σταθερή ακολουθία συγκρίσεων με βάση το μεσαίο στοιχείο.

Αναζήτηση παρεμβολής

Απόδοση μέσης περίπτωσης:

Η αναζήτηση παρεμβολής έχει μέση χρονική πολυπλοκότητα περίπτωσης O(loglogn), υποθέτοντας ότι τα δεδομένα είναι ομοιόμορφα κατανεμημένα.

επίδραση της κατανομής των δεδομένων:

Η απόδοση της αναζήτησης παρεμβολής είναι ιδιαίτερα ευαίσθητη στην κατανομή του συνόλου δεδομένων:

Ομοιόμορφη κατανομή:

Συχνά επιτυγχάνει τη μέση χρονική πολυπλοκότητα περίπτωσής της της τάξης του O(loglogn). Αυτό συμβαίνει επειδή ο τύπος παρεμβολής προβλέπει με ακρίβεια τη θέση του κλειδιού αναζήτησης.

Μη ομοιόμορφη κατανομή:

Για δεδομένα που δεν είναι ομοιόμορφα κατανεμημένα, η απόδοση μπορεί να υποβαθμιστεί σημαντικά. Εάν τα δεδομένα είναι συγκεντρωμένα ή ακολουθούν λοξή κατανομή, η εκτιμώμενη θέση μπορεί να απέχει πολύ από την πραγματική θέση, οδηγώντας σε περισσότερες συγκρίσεις και συνεπώς σε χειρότερη απόδοση. Στη χειρότερη περίπτωση, μπορεί να υποβαθμιστεί σε O(n), παρόμοια με μια γραμμική αναζήτηση, ειδικά σε περιπτώσεις όπου το σύνολο δεδομένων έχει ακραίες τιμές ή ακραίες τιμές.

Screenshot εκτέλεσης άσκησης:

```
Give Range Start: 10000
Give Range End: 30000
---------------- THE REGIONS in that Range of births are:---------
--With Binary Search:--
Gisborne region: 13041
Southland region: 21960
Taranaki region: 27327

--With Interpolation Search:--
Gisborne region: 13041
Southland region: 21960
Taranaki region: 27327

-------------Durations:--------------
Duration of Binary Search: 368 clock ticks
Duration of Interpolation Search: 697 clock ticks
```

4. Δυική Αναζήτηση Παρεμβολής (BIS)

Η δυαδική αναζήτηση παρεμβολής είναι μια βελτιωμένη εκδοχή της δυαδικής αναζήτησης, η οποία εκμεταλλεύεται την κατανομή των δεδομένων για να πραγματοποιήσει πιο αποδοτικές αναζητήσεις. Αντί να εξετάζει το μεσαίο στοιχείο του πίνακα όπως η δυαδική αναζήτηση, η δυαδική αναζήτηση παρεμβολής υπολογίζει τη θέση όπου είναι πιο πιθανό να βρίσκεται το στοιχείο-κλειδί, χρησιμοποιώντας έναν τύπο παρεμβολής.

Χωρίς Βελτίωση

Χρονική Πολυπλοκότητα Μέσης Περίπτωσης

Η χρονική πολυπλοκότητα της μέσης περίπτωσης της δυαδικής αναζήτησης παρεμβολής είναι *O(loglogn)*, όπου *n* είναι ο αριθμός των στοιχείων του πίνακα. Αυτή η πολυπλοκότητα είναι για πίνακες όπου τα στοιχεία είναι ομοιόμορφα κατανεμημένα, καθώς επιτρέπει στην αναζήτηση να πλησιάζει το στοιχείο-κλειδί πολύ πιο γρήγορα.

Χρονική Πολυπλοκότητα Χειρότερης Περίπτωσης

Η χρονική πολυπλοκότητα της χειρότερης περίπτωσης της δυαδικής αναζήτησης παρεμβολής είναι $O(\sqrt{n})$. Αυτό συμβαίνει όταν τα στοιχεία του πίνακα δεν είναι ομοιόμορφα κατανεμημένα και η παρεμβολή αποτυγχάνει να πλησιάσει αποτελεσματικά το στοιχείο-κλειδί.

Με Βελτίωση

Χρονική Πολυπλοκότητα Μέσης Περίπτωσης

Η χρονική πολυπλοκότητα της μέσης περίπτωσης δεν επηρεάζεται απο την βελτίωση την εκθετικής αύξησης των αλμάτων αναζήτησης. Αρα παραμένει O(loglogn).

Χρονική Πολυπλοκότητα Χειρότερης Περίπτωσης

Με την εκθετική αύξηση των αλμάτων της αναζήτησης η πολυπλοκότητα γίνεται O(logn). Αυτό ισχύει αφου γίνονται ολοένα και μεγαλύτερα άλματα. Οπότε, το τελευταίο υποδιάστημα θα είναι πολύ μεγαλύτερο από $\sqrt{n}$.

Screenshot εκτέλεσης άσκησης:

```
Give Range Start: 30000
Give Range End: 50000
---------------- THE REGIONS in that Range of births are:---------
----Without Optimization:----
Hawke's Bay region: 38973
Northland region: 40191
Otago region: 40845
----With Optimization:----
Hawke's Bay region: 38973
Northland region: 40191
Otago region: 40845
Duration of BIS with no optimization: 4620 clock ticks
Duration of BIS with optimization: 3260 clock ticks
```

Πίνακας χρόνων για πολλαπλές εκτελέσεις:

Αλγόριθμος	**Start = 30000 End = 50000**	**Start = 1000 End = 10000**	**Start = 20000 End = 50000**	**Start = 20000 End = 30000**
Χωρίς Βελτίωση	4620	3684	1003	1612
Με Βελτίωση	3260	1875	951	975

Παρατηρούμε πως οι χρόνοι της βελτιωμένης έκδοσης είναι μικρότεροι για κάθε αναζήτηση που εκτελούμε.

PART II: “BSTs && HASHING”

(A): Το ΔΔΑ διατάσσεται ως προς το πεδίο Region

(1): Απεικόνιση του ΔΔΑ με ενδο-διατεταγμένη διάσχιση

Ακολουθεί παράδειγμα εκτέλεσης προγράμματος για το ερώτημα 1:

```
Load data on Binary Search Tree or Hash table? [bst/hash]: bst
Binary Search Tree key? [region/births]: region
What do you want to do? [show/search/modify/delete/exit/help]: show
Canterbury region: 125601
Marlborough region: 9147
New Zealand: 1083162
Nelson region: 9861
Manawatu-Wanganui region: 55497
Hawke's Bay region: 38973
Gisborne region: 13041
Bay of Plenty region: 70983
Auckland region: 391893
Region not stated or area outside region: 792
Southland region: 21960
Otago region: 40845
Tasman region: 8805
Taranaki region: 27327
West Coast region: 6747
Wellington region: 111969
Waikato region: 109491
Northland region: 40191
```

Στην παραπάνω εικόνα φαίνονται τυπωμένες όλες οι περιοχές και ο αριθμός γεννήσεων που τους αντιστοιχεί με σειρά ενδο-διατεταγμένη διάσχιση.

(2): Αναζήτηση αριθμού γεννήσεων με βάση χρονική περίοδο και περιοχή

```
What do you want to do? [show/search/modify/delete/exit/help]: search
Type Region name to search: Gisborne region
Type Period [2005-2022]: 2010
Births in Gisborne region on 2010 are: 783
```

Στην παραπάνω εικόνα φαίνεται η αναζήτηση αριθμού γεννήσεων μιας περιοχής παίρνωντας ως είσοδο από τον χρήστη το όνομα της περιοχής και την χρονολογία.

(3): Τροποποίηση του πεδίου αριθμού γεννήσεων για συγκεκριμένη χρονική περίοδο και περιοχή

```
What do you want to do? [show/search/modify/delete/exit/help]: modify
Type Region name to modify: Gisborne region
Type Period [2005-2022]: 2010
Type new Birth cout for year 2010: 420

What do you want to do? [show/search/modify/delete/exit/help]: search
Type Region name to search: Gisborne region
Type Period [2005-2022]: 2010
Births in Gisborne region on 2010 are: 420
```

Στην παραπάνω εικόνα φαίνεται η διαδικασία της τροποποίησης του αριθμού γεννήσεων μιας περιοχής για μια συγκεκριμένη χρονολογία, τα οποία δίνονται από τον χρήστη. Στη συνέχεια πραγματοποιούμε αναζήτηση για την συγκεκριμενη περιοχη και χρονολογια που επιλέχθηκε και βλέπουμε πως όντως έχουν γίνει οι αλλαγές.

(4): Διαγραφή μιας εγγραφής βάσει της περιοχής

```
What do you want to do? [show/search/modify/delete/exit/help]: delete
Type Region name to delete: Gisborne region

What do you want to do? [show/search/modify/delete/exit/help]: search
Type Region name to search: Gisborne region
Region Not Found!
```

Στην παραπάνω εικόνα φαίνεται η διαγραφή μιας εγγραφής με βάσει το όνομα της περιοχής το οποίο έχει δοθεί από τον χρήστη. Στη συνέχεια πραγματοποιείται αναζήτηση της περιοχής που διαγράφτηκε και τυπώνεται ως αποτέλεσμα “Region Not Found”.

(B): Το ΔΔΑ διατάσσεται ως προς το πεδίο Count_of_Births

Η διαφορά με το ερώτημα Α είναι στην εντολή makeTree(nodeVector, "births") στην οποία το “births” δηλώνει πως το δέντρο θα διαταχθεί ως προς τις συνολικές γεννήσεις της κάθε περιοχής αντί ως προς το όνομα της περιοχής που θα γινόταν με την παράμετρο “region”.

(1): Εύρεση περιοχής με τον ελάχιστο αριθμό γεννήσεων

```
Load data on Binary Search Tree or Hash table? [bst/hash]: bst
Binary Search Tree key? [region/births]: births
What do you want to do? [min/max/exit/help]: min
Region with least total births: Region not stated or area outside region with 792 total births.
```

Στην παραπάνω εικόνα φαίνεται πως ο χρήστης επιλέγει το min και το πρόγραμμα τυπώνει την περιοχή με τον ελάχιστο αριθμό γεννήσεων. Για την εύρεση του ελαχίστου αριθμού γεννήσεων χρησιμοποιείται ένας αναδρομικός αλγόριθμος ο οποίος ακολουθεί το πιο αριστερό μονοπάτι του δέντρου μέχρι να φτάσει στο φυλλο.

(2): Εύρεση περιοχής με τον μέγιστο αριθμό γεννήσεων

```
Load data on Binary Search Tree or Hash table? [bst/hash]: bst
Binary Search Tree key? [region/births]: births
What do you want to do? [min/max/exit/help]: max
Region with most total births: New Zealand with 1083162 total births.
```

Στην παραπάνω εικόνα φαίνεται πως ο χρήστης επιλέγει το max και το πρόγραμμα τυπώνει την περιοχή με τον μέγιστο αριθμό γεννήσεων. Για την εύρεση του μέγιστου αριθμού γεννήσεων χρησιμοποιείται ένας αναδρομικός αλγόριθμος ο οποίος ακολουθεί το πιο δεξί μονοπάτι του δέντρου μέχρι να φτάσει στο φυλλο.

(Γ): Υλοποίηση (Α) κάνοντας χρήση HASHING με αλυσίδες

Εδώ κάνουμε χρήση hashing με αλυσιδες *(linked list)*. Η συνάρτηση κατακερματισμού είναι η: *hash(key) = Sum(ASCII(k_i)) mod 11*, με k_i = {key[0], key[1], … , k[n-1]}, για key: string μεγέθους n. Άρα ο αριθμός των θέσεων του πίνακα κατακερματισμού ειναι 11.

(1): Αναζήτηση αριθμού γεννήσεων με βάση χρονική περίοδο και περιοχή

Screenshot εκτέλεσης άσκησης:

```
Load data on Binary Search Tree or Hash table? [bst/hash]: hash
What do you want to do? [search/modify/delete/exit/help]: search
Give Region name to search: Southland region
Give Period to search: 2012
Southland region: 1245
```

(2): Τροποποίηση του πεδίου αριθμού γεννήσεων για συγκεκριμένη χρονική περίοδο και περιοχή

Screenshot εκτέλεσης άσκησης:

```
What do you want to do? [search/modify/delete/exit/help]: modify
Give Region name to modify: Southland region
Give Period to modify: 2012
Give new Value: 1010

What do you want to do? [search/modify/delete/exit/help]: search
Give Region name to search: Southland region
Give Period to search: 2012
Southland region: 1010
```

Στην παραπάνω εικόνα φαίνεται η διαδικασία της τροποποίησης του αριθμού γεννήσεων μιας περιοχής για μια συγκεκριμένη χρονολογία, τα οποία δίνονται από τον χρήστη. Στη συνέχεια πραγματοποιούμε αναζήτηση για την συγκεκριμενη περιοχη και χρονολογια που επιλέχθηκε και βλέπουμε πως όντως έχουν γίνει οι αλλαγές.

(3): Διαγραφή μιας εγγραφής βάσει της περιοχής

Screenshot εκτέλεσης άσκησης:

```
What do you want to do? [search/modify/delete/exit/help]: delete
Give region name to delete: Southland region

What do you want to do? [search/modify/delete/exit/help]: search
Give Region name to search: Southland region
Give Period to search: 2012
Region Southland region not found!
```

Στην παραπάνω εικόνα φαίνεται η διαγραφή μιας εγγραφής με βάσει το όνομα της περιοχής το οποίο έχει δοθεί από τον χρήστη. Στη συνέχεια πραγματοποιείται αναζήτηση της περιοχής που διαγράφτηκε και τυπώνεται ως αποτέλεσμα “Region Not Found”.

ΣΗΜΕΙΩΣΗ1:

Για την αποδοτικότερη οργάνωση των δεδομένων στα ερωτήματα (Α) και (Β) επιλέξαμε την χρήση λίστας (*list<Node> nodeVecto*r). Αυτό μας χρησιμεύει κυρίως στο υποερώτημα (4) του (Α). Επειδή με την λίστα οι κόμβοι του δέντρου δεν βρίσκονται σε διαδοχικές θέσεις μνήμης, κατά την διαγραφή ενός κόμβου δεν επηρεάζονται οι θέσεις των επόμενων κόμβων της λίστας. Στην περίπτωση που χρησιμοποιούσαμε πίνακα, με την διαγραφή ενός κόμβου, όλοι οι επόμενοι κόμβοι θα άλλαζαν θέση στην μνήμη και οι pointers που θα έδειχναν στα παιδιά και στους γονείς του κάθε κόμβου θα έδειχναν σε λάθος θέση. Για αυτό θα χρειαζόταν να μετατοπίσουμε τους pointers να δείχνουν σε μια θέση μικρότερη, το οποίο και θα κόστιζε σε χρόνο.

Στο ερώτημα (Γ) χρησιμοποιούμε πίνακα για τον πίνακα κατακερματισμού (*list<Region> hashTable[M]*) μεγέθους Μ (= τιμή κατακερματισμού). Αφού ο πίνακας θα έχει πάντα σταθερό μέγεθος και χρειαζόμαστε τυχαία προσπέλαση με σταθερό χρόνο η δομή πίνακα είναι ιδανική. Παρόλα αυτά, κάθε θέση του πίνακα αποτελείται από μια λίστα (*list<Region>*) αφού χρειαζόμαστε δυναμική διαχείριση της μνήμης για τις διαγραφές των έγγραφων του υποερωτήματος (3). Για να ειναι αποδοτική η αποθήκευση του πίνακα κατακερματισμού θα πρέπει το μέγεθος Μ να μην είναι πολύ μικρό, ώστε να μειωθούν οι συγκρούσεις και αρα ο χρόνος προσπέλασης, αλλα ούτε μεγάλο, ώστε να μήν καταναλώνεται χώρος στην μνήμη από κενές θέσεις του πίνακα.

Οδηγίες Εκτέλεσης

Το αρχείο των δεδομένων θα πρέπει να έχει το όνομα *bd-dec22-births-deaths-by-region.csv* και να βρίσκεται στον ίδιο φάκελο με το εκτελέσιμο πρόγραμμα.
Μαζί με τα αρχεία που ζητούνται, παραθέτουμε και ένα makefile αρχείο για την πιο εύκολη εκτέλεση των προγραμμάτων. Παρακάτω ακολουθούν οι εντολές.

Part I.1

```
make part11
./partI.1.out
```

ή εναλλακτικά χώρις το make:

```
g++ -c basicFunctions.cpp
g++ -c sortingFunctions.cpp
g++ -c PartI.1.cpp
g++ -o partI.1.out basicFunctions.o sortingFunctions.o PartI.1.o
./partI.1.out
```

Part I.2

```
make part12
./partI.2.out
```

ή εναλλακτικά χώρις το make:

```
g++ -c basicFunctions.cpp
g++ -c sortingFunctions.cpp
g++ -c PartI.2.cpp
g++ -o partI.2.out basicFunctions.o sortingFunctions.o PartI.2.o
./partI.2.out
```

Part I.3

```
make part13
./partI.3.out
```

ή εναλλακτικά χώρις το make:

```
g++ -c basicFunctions.cpp
g++ -c sortingFunctions.cpp
g++ -c searchFunctions.cpp
g++ -c PartI.3.cpp
g++ -o partI.3.out searchFunctions.o basicFunctions.o sortingFunctions.o PartI.3.o
./partI.3.out
```

Part I.4

```
make part14
./partI.4.out
```

ή εναλλακτικά χώρις το make:

```
g++ -c basicFunctions.cpp
g++ -c sortingFunctions.cpp
g++ -c searchFunctions.cpp
g++ -c PartI.4.cpp
g++ -o partI.4.out searchFunctions.o basicFunctions.o sortingFunctions.o PartI.4.o
./partI.4.out
```

Part II

```
make part2
./partII.out
```

ή εναλλακτικά χώρις το make:

```
g++ -c basicFunctions.cpp
g++ -c sortingFunctions.cpp
g++ -c searchFunctions.cpp
g++ -c PartII.B.cpp
g++ -c PartII.A.cpp
g++ -c PartII.C.cpp
g++ -c PartII.cpp
g++ -o partII.out PartII.C.o PartII.B.o PartII.A.o searchFunctions.o basicFunctions.o
./partII.out
```