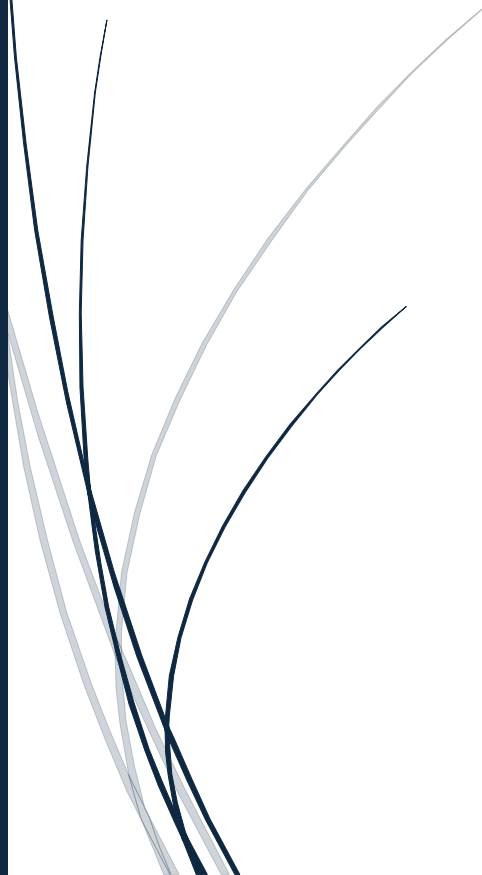




2024

ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ

Απαντήσεις Project



Απόστολος Ζεκυριάς (1100554)
Σπυρίδων Μανταδάκης (1100613)
Παναγιώτης Παπανικολάου (1104804)
Αλέξανδρος Γεώργιος Χαλαμπάκης (1100754)

Γενικές Πληροφορίες

Σχόλια: Κατά την ανάπτυξη της γραπτής αναφοράς, προσθέσαμε τον πηγαίο κώδικα κάθε προγράμματος που υλοποιήσαμε, διότι δεν ήμασταν σίγουροι για τα περιεχόμενα της αναφοράς μας.

Κατά την ανάπτυξη των προγραμμάτων, δεν αντιμετωπίσαμε κάποιο σοβαρό πρόβλημα. Ωστόσο, δυσκολευτήκαμε στην υλοποίηση της δυαδικής αναζήτησης με Παρεμβολή. Δεν είμαστε 100% σίγουροι για τη λειτουργικότητα και αν είναι σωστή η κατασκευή του κώδικα.

Άλλη μία δυσκολία που αντιμετωπίσαμε ήταν στο 2ο μέρος της εργασίας, στο Β) Ερώτημα, όπου το μενού ζητάει να εμφανίζει ελάχιστο και μέγιστο count για εγγραφή. Δυσκολευτήκαμε στην κατανόηση, διότι το ζητούμενο είχε λίγη ασάφεια. Υλοποιήσαμε όλα τα ζητούμενα της εργαστηριακής άσκησης εκτός από τα bonus, τα οποία δεν προλάβαμε.

Στις επόμενες σελίδες παρουσιάζονται οι απαντήσεις της ομάδας μας στο Project του μαθήματος "**Δομές Δεδομένων**". Σε αυτήν τη σελίδα παρέχονται πληροφορίες σχετικά με τα μέλη της ομάδας.

Η ομάδα αποτελείται από τους εξής φοιτητές:

Απόστολος Ζεκυριάς

Σπυρίδων Μανταδάκης

Παναγιώτης Παπανικολάου

Αλέξανδρος Γεώργιος Χαλαμπάκης

Αναλυτικότερες Πληροφορίες:

<p>Απόστολος Ζεκυριάς 1100554</p> <p>up1100554@ac.upatras.gr</p> <p>Φοιτητής 2ου έτους</p>	<p>Σπυρίδων Μανταδάκης 1100613</p> <p>up1100613@ac.upatras.gr</p> <p>Φοιτητής 2ου έτους</p>	<p>Παναγιώτης Παπανικολάου 1104804</p> <p>up1104804@ac.upatras.gr</p> <p>Φοιτητής 2ου έτους</p>	<p>Αλέξανδρος Γεώργιος Χαλαμπάκης 1100754</p> <p>up1100754@ac.upatras.gr</p> <p>Φοιτητής 2ου έτους</p>
---	--	--	---

Περιεχόμενα

PART I: “Sorting and Searching Algorithms”:

1. Διαδικασία Ανάπτυξης πρώτου μέρους.....	3
2. Merge Sort και Quick Sort.....	9
3. Heap Sort και Counting Sort.....	18
4. Δυαδική Αναζήτηση και Αναζήτηση με Παρεμβολή.....	26
5. Δυαδική Αναζήτηση Παρεμβολή (BIS).....	35

PART II: “BSTs & HASHING”:

A. Δυαδικό Δένδρο Αναζήτησης (ΔΔΑ).....	40
B. Τροποποίηση ΔΔΑ.....	49
Γ. HASHING.....	55
Ενοποίηση A,B,Γ.....	62

1. Διαδικασία Ανάπτυξης πρώτου μέρους

Για την ανάπτυξη του πρώτου μέρους της εργασίας ξεκινήσαμε με την ανάγνωση του CSV αρχείου που περιέχει τα δεδομένα που θέλουμε να διαβάσουμε. Φτιάξαμε ένα header file (“`read_print.h`”) που περιέχει μέσα τη δομή για τα στοιχεία κάθε γραμμής μέσα στο αρχείο και το array of structs, καθώς και τις συναρτήσεις για ανάγνωση των περιεχομένων του αρχείου και την εκτύπωσή τους.

Φτιάξαμε και ένα C++ αρχείο (“`FileToStruct.cpp`”) που εμπεριέχει το παραπάνω header αρχείο. Το αρχείο “`FileToStruct.cpp`” το χρησιμοποιούμε για να ελέγξουμε αν η υλοποίηση είναι σωστή και δεν μας δημιουργούνται προβλήματα στο τρέξιμο του κώδικα.

Παρακάτω φαίνεται ο κώδικας του αρχείου “`FileToStruct.cpp`” και η τελική μορφή του header file.

FileToStruct.cpp:

```
#include "read_print.h" //access read_print.h
#include <iostream>
#include <string>

using namespace std;

void Print_Data()
{
    //for loop to print array of structs data
    for (int i = 0; i < row_counter; ++i) {
        cout << "Period: " << data[i].Period << ", Birth_Death: " <<
(data[i].Birth_Death ? "Birth" : "Death") << ", Region: " << data[i].Region << ",
Count: " << data[i].Count << endl;
    }
}

int main()
{
    Read_Data(); //access Read_Data() from read_print.h
```

```

    Print_Data(); //access Print_Data() from read_print.h

    return 0;
}

```

read_print.h:

```

#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
using namespace std;

struct Row {
    int Period;
    bool Birth_Death;    //0=Death , 1=Birth
    string Region;
    int Count;
};

const int MAXROWS = 648; //max number of rows
const int MAXSUMS = 17; //max number of summedCount Regions

Row data[MAXROWS]; //make array of Row structs

int row_counter=0;

struct SummedCount {
    string Region;
    int Sum;
};

SummedCount Summedcounts[MAXSUMS]; // Make array of summed count Regions

int Read_Data()
{

```

```

//open the file and check for errors
ifstream inputFile("bd-dec22-births-deaths-by-region.txt");
if (!inputFile) {
    cerr << "Error opening file." << endl;
    return 1;
}

string line;

bool FirstLineSkipped = false; // Flag to track whether the first line has
been skipped

while (getline(inputFile, line) && row_counter < MAXROWS) {
    if (!FirstLineSkipped) {
        FirstLineSkipped = true;
        continue; // skip first line
    }

    stringstream ss(line); //read and write in string line
    string token; // data from file to be read
    string tokens[4];

    int index = 0; //number of commas to be read

    while (getline(ss, token, ',')) { //read row of file and return
data(token) until it reaches a comma
        tokens[index++] = token; //store data from between the commas of
each row in tokens[]
        if (index >= 4) {
            break; // Avoid accessing out of bounds
        }
    }

    if (index != 4) {
        cerr << "Error: Invalid data format in line: " << line << endl;
        continue; //incorrect amount of commas
    }

    try {
        //store read data into the array of structs
        data[row_counter].Period = stoi(tokens[0]);
        data[row_counter].Birth_Death = (tokens[1] == "Births"); // Set true
if "Births", false otherwise
    }
}

```

```

        data[row_counter].Region = tokens[2];
        data[row_counter].Count = stoi(tokens[3]);
    } catch (const exception &e) {
        cerr << "Error: Invalid integer conversion in line: " << line <<
endl;
        continue; //catch errors when converting to integers
    }

    row_counter++; //go to the next row

}

inputFile.close(); //close file

}

// Function to calculate birth sums for each region between 2005 and 2022
void CalculateBirthSums() {

    // Initialize summed counts for each region to zero
    for (int i = 0; i < MAXSUMS; ++i) {
        Summedcounts[i].Region = data[i].Region;
        Summedcounts[i].Sum = 0;
    }

    // Calculate summed counts for each region between 2005 and 2022
    for (int i = 0; i < row_counter; ++i) {
        if (data[i].Period >= 2005 && data[i].Period <= 2022 &&
data[i].Birth_Death) { // if data[1].Birth_Detah == 1
            // Find the index of the region in Summedcounts array
            int regionIndex = -1;
            for (int j = 0; j < MAXROWS; ++j) {
                if (Summedcounts[j].Region == data[i].Region) { //make a copies
of the regions in data[] that have births
                    regionIndex = j;
                    break;
                }
            }
            // Add birth count to the summed count for the corresponding region
            if (regionIndex != -1) {
                Summedcounts[regionIndex].Sum += data[i].Count;
            }
        }
    }
}

```

```

    }
}

// Function to calculate death sums for each region between 2005 and 2022
void CalculateDeathSums() {

    // Initialize summed counts for each region to zero
    for (int i = 0; i < MAXSUMS; ++i) {
        Summedcounts[i].Region = data[i].Region;
        Summedcounts[i].Sum = 0;
    }

    // Calculate summed counts for each region between 2005 and 2022
    for (int i = 0; i < row_counter; ++i) {
        if (data[i].Period >= 2005 && data[i].Period <= 2022 &&
(!data[i].Birth_Death)) { // if data[1].Birth_Detah == 0
            // Find the index of the region in Summedcounts array
            int regionIndex = -1;
            for (int j = 0; j < MAXROWS; ++j) {
                if (Summedcounts[j].Region == data[i].Region) { //make a copies
of the regions in data[] that have deaths
                    regionIndex = j;
                    break;
                }
            }
            // Add death count to the summed count for the corresponding region
            if (regionIndex != -1) {
                Summedcounts[regionIndex].Sum += data[i].Count;
            }
        }
    }
}

void PrintSummedCounts(SummedCount Summedcounts[], int size) {
    for (int i = 0; i < size; ++i) {
        cout << "Region: " << Summedcounts[i].Region << ", Sum: " <<
Summedcounts[i].Sum << endl;
    }
}

```


Παρακάτω φαίνεται το στιγμιότυπο που περιέχει ένα μέρος του output του “FileToStruct.cpp”.

```
Period: 2005, Birth_Death: Birth, Region: Northland region, Count: 2067
Period: 2005, Birth_Death: Birth, Region: Auckland region, Count: 20745
Period: 2005, Birth_Death: Birth, Region: Waikato region, Count: 5667
Period: 2005, Birth_Death: Birth, Region: Bay of Plenty region, Count: 3771
Period: 2005, Birth_Death: Birth, Region: Gisborne region, Count: 777
Period: 2005, Birth_Death: Birth, Region: Hawke's Bay region, Count: 2115
Period: 2005, Birth_Death: Birth, Region: Taranaki region, Count: 1410
Period: 2005, Birth_Death: Birth, Region: Manawatu-Wanganui region, Count: 3093
Period: 2005, Birth_Death: Birth, Region: Wellington region, Count: 6225
Period: 2005, Birth_Death: Birth, Region: Tasman region, Count: 513
Period: 2005, Birth_Death: Birth, Region: Nelson region, Count: 519
Period: 2005, Birth_Death: Birth, Region: Marlborough region, Count: 471
Period: 2005, Birth_Death: Birth, Region: West Coast region, Count: 336
Period: 2005, Birth_Death: Birth, Region: Canterbury region, Count: 6603
Period: 2005, Birth_Death: Birth, Region: Otago region, Count: 2151
Period: 2005, Birth_Death: Birth, Region: Southland region, Count: 1212
Period: 2005, Birth_Death: Birth, Region: "Region not stated or area outside region", Count: 57
Period: 2005, Birth_Death: Birth, Region: New Zealand, Count: 57747
Period: 2005, Birth_Death: Death, Region: Northland region, Count: 1167
Period: 2005, Birth_Death: Death, Region: Auckland region, Count: 6873
Period: 2005, Birth_Death: Death, Region: Waikato region, Count: 2520
Period: 2005, Birth_Death: Death, Region: Bay of Plenty region, Count: 1956
```

Merge Sort και Quick Sort

Για την υλοποίηση του πρώτου ερωτήματος ξεκινήσαμε φτιάχνοντας ένα c++ αρχείο με όνομα “mergesort.cpp”. Ακολουθήσαμε τα βήματα του αλγόριθμου και φτιάξαμε το παρακάτω πρόγραμμα.

```
#include "read_print.h" // Access read_print.h
#include <iostream>
#include <string>
#include <chrono> // Include chrono for time measurements

using namespace std;
using namespace chrono;

// Merge function to merge two sorted arrays
void merge(SummedCount arr[], int l, int m, int r) {
    int n1 = m - l + 1; // Number of elements in the left subarray
    int n2 = r - m; // Number of elements in the right subarray

    // Create temporary arrays
    SummedCount L[n1], R[n2];
```

```

// Copy data to temporary arrays L[] and R[]
for (int i = 0; i < n1; i++)
    L[i] = arr[l + i];
for (int j = 0; j < n2; j++)
    R[j] = arr[m + 1 + j];

// Merge the temporary arrays back into arr[l..r]
int i = 0; // Initial index of the first subarray
int j = 0; // Initial index of the second subarray
int k = l; // Initial index of the merged subarray
while (i < n1 && j < n2) {
    if (L[i].Sum <= R[j].Sum) {
        arr[k] = L[i];
        i++;
    } else {
        arr[k] = R[j];
        j++;
    }
    k++;
}

// Copy the remaining elements of L[], if any
while (i < n1) {
    arr[k] = L[i];
    i++;
    k++;
}

// Copy the remaining elements of R[], if any
while (j < n2) {
    arr[k] = R[j];
    j++;
    k++;
}
}

// Main function of merge sort
void mergeSort(SummedCount arr[], int l, int r) {
    if (l < r) {
        // Find the middle point to divide the array into two halves
        int m = l + (r - l) / 2;

        // Call mergeSort for the first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
    }
}

```

```

        // Merge the two halves sorted in the previous steps
        merge(arr, l, m, r);
    }
}

int main()
{
    Read_Data(); // Call Read_Data() to read the data

    CalculateBirthSums(); // Calculate and store summed birth counts for each
region between 2005 and 2022

    // Start measuring time
    auto start_time = high_resolution_clock::now();

    const int iterations = 1000; // Number of iterations to find average
execution time of the algorithm
    for (int i = 0; i < iterations; ++i) {

        // Sort Summedcounts based on the total sum of each region in ascending
order using MergeSort
        mergeSort(Summedcounts, 0, MAXSUMS - 1);
    }

    // Stop measuring time
    auto end_time = high_resolution_clock::now();

    // Calculate the duration of the sorting
    auto duration = duration_cast<microseconds>(end_time - start_time); //
Calculate the duration in microseconds

    // Print the contents of the Summedcounts array
    cout << "Summed Birth Counts for each region between 2005 and 2022 :" << endl
<< endl;
    PrintSummedCounts(Summedcounts, MAXSUMS);

    // Print the average execution time of merge sort
    cout << "Average MergeSort execution time: " << duration.count() / iterations
<< " microseconds" << endl;

    return 0;
}

```

Αυτός ο κώδικας υλοποιεί τη συγχώνευση (merge sort) για την ταξινόμηση πίνακα δομών “SummedCount”, ο οποίος περιέχει στοιχεία που αναπαριστούν άθροισμα και μέτρηση γεννήσεων για κάθε περιοχή και έτος μεταξύ 2005 και 2022.

Βιβλιοθήκες και ονοματοχώροι: Ο κώδικας χρησιμοποιεί βιβλιοθήκες όπως `<iostream>`, `<string>` για την επεξεργασία εισόδου/εξόδου και `<chrono>` για τη μέτρηση του χρόνου.

Δομή “SummedCount”: Περιέχει τουλάχιστον δύο μέλη: Sum (το σύνολο των γεννήσεων) και Count (ο αριθμός των γεννήσεων).

merge(): Η συνάρτηση merge() εκτελεί τη συγχώνευση δύο ταξινομημένων υποπινάκων σε έναν ταξινομημένο υποπίνακα. Χρησιμοποιείται στη συγχώνευση δύο ημιδιαχωρισμένων υποπινάκων κατά τη διάρκεια της merge sort.

mergeSort(): Η συνάρτηση mergeSort() είναι η κύρια συνάρτηση του αλγορίθμου merge sort, η οποία διαιρεί αναδρομικά τον πίνακα σε μικρότερα κομμάτια και καλεί την merge() για τη συγχώνευση τους.

main(): Στη συνάρτηση main(), γίνεται κλήση της Read_Data() για την ανάγνωση δεδομένων, της CalculateBirthSums() για τον υπολογισμό των συνολικών αθροισμάτων γεννήσεων για κάθε περιοχή, και στη συνέχεια γίνεται η ταξινόμηση των δομών SummedCount χρησιμοποιώντας τον merge sort.

Μέτρηση χρόνου: Ο χρόνος εκτέλεσης του merge sort μετριέται χρησιμοποιώντας το `<chrono>`, και υπολογίζεται ο μέσος όρος του χρόνου εκτέλεσης από 1000 επαναλήψεις.

Έξοδος: Τα αποτελέσματα της ταξινόμησης εκτυπώνονται με την PrintSummedCounts(), ενώ ο μέσος χρόνος εκτέλεσης του αλγορίθμου merge sort εκτυπώνεται επίσης στην έξοδο.

Summed Birth Counts for each region between 2005 and 2022 :

```
Region: "Region not stated or area outside region", Sum: 792
Region: West Coast region, Sum: 6747
Region: Tasman region, Sum: 8805
Region: Marlborough region, Sum: 9147
Region: Nelson region, Sum: 9861
Region: Gisborne region, Sum: 13041
Region: Southland region, Sum: 21960
Region: Taranaki region, Sum: 27327
Region: Hawke's Bay region, Sum: 38973
Region: Northland region, Sum: 40191
Region: Otago region, Sum: 40845
Region: Manawatu-Wanganui region, Sum: 55497
Region: Bay of Plenty region, Sum: 70983
Region: Waikato region, Sum: 109491
Region: Wellington region, Sum: 111969
Region: Canterbury region, Sum: 125601
Region: Auckland region, Sum: 391893
Average MergeSort execution time: 33 microseconds
```

Για την υλοποίηση της Quicksort φτιαξαμε ενα c++ αρχαιο με ονομα quicksort.cpp. Ακολουθησαμε τα βηματα του αλγοριθμου και φτιαξαμε το παρακάτω πρόγραμμα.

```
#include "read_print.h" // Access read_print.h
#include <iostream>
#include <string>
#include <chrono> // Include chrono for time measurements

using namespace std;
using namespace chrono;

// Partition function for QuickSort
int partition(SummedCount arr[], int low, int high) {
    SummedCount pivot = arr[high]; // Choose the pivot element
    int i = low - 1; // Index of smaller element

    for (int j = low; j < high; j++) {
```

```

        // If current element is smaller than or equal to pivot
        if (arr[j].Sum <= pivot.Sum) {
            i++; // Increment index of smaller element
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return i + 1;
}

// Main QuickSort function
void quickSort(SummedCount arr[], int low, int high) {
    if (low < high) {
        // pi is partitioning index
        int pi = partition(arr, low, high);

        // Recursively sort elements before and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

int main() {
    Read_Data(); // Access Read_Data() from read_print.h

    CalculateBirthSums(); // Calculate and store summed birth counts for each
region between 2005 and 2022

    // Start measuring time
    auto start_time = high_resolution_clock::now();

    const int iterations = 1000; // Number of iterations to find average
execution time of the algorithm
    for (int i = 0; i < iterations; ++i) {

        // Sort Summedcounts based on the total sum of each region using
QuickSort
        quickSort(Summedcounts, 0, MAXSUMS - 1);
    }

    // Stop measuring time
    auto end_time = high_resolution_clock::now();

    // Calculate duration of the sorting

```

```

    auto duration = duration_cast<microseconds>(end_time - start_time); //
Calculate the duration in microseconds

    // Print the contents of Summedcounts array
    cout << "Summed Birth Counts for each region between 2005 and 2022 :" << endl
<< endl;
    PrintSummedCounts(Summedcounts, MAXSUMS);

    // Print the average execution time of quick sort
    cout << "Average Quicksort execution time: " << duration.count() / iterations
<< " microseconds" << endl;

    return 0;
}

```

Αυτός ο κώδικας υλοποιεί τον αλγόριθμο QuickSort για την ταξινόμηση του πίνακα δομών “SummedCount”, ο οποίος περιέχει στοιχεία που αναπαριστούν άθροισμα και μέτρηση γεννήσεων για κάθε περιοχή και έτος μεταξύ 2005 και 2022.

Βιβλιοθήκες και ονοματοχώροι: Ο κώδικας χρησιμοποιεί τις βιβλιοθήκες <iostream>, <string> για την επεξεργασία εισόδου/εξόδου και <chrono> για τη μέτρηση του χρόνου.

Δομή “SummedCount”: Περιέχει τουλάχιστον ένα πεδίο Sum, το οποίο χρησιμοποιείται για τη σύγκριση και την ταξινόμηση των δεδομένων.

partition(): Η συνάρτηση partition() δέχεται ως είσοδο έναν πίνακα “arr” και δύο δείκτες low και high που υποδεικνύουν τα όρια του τμήματος του πίνακα που θα γίνει διαίρεση (partition). Χρησιμοποιείται για να διαχωρίσει τα στοιχεία μιας δομής “SummedCount” με βάση το πεδίο “Sum”.

quickSort(): Η κύρια συνάρτηση quickSort() εκτελεί την αναδρομική διαδικασία του QuickSort. Αρχικά καλεί την partition() για να χωρίσει τον πίνακα σε δύο υποπίνακες, και στη συνέχεια καλεί τον εαυτό της αναδρομικά για την ταξινόμηση των δύο νέων υποπινάκων.

main(): Στη συνάρτηση main(), γίνεται κλήση της Read_Data() για την ανάγνωση δεδομένων, και της CalculateBirthSums() για τον υπολογισμό των συνολικών αθροισμάτων γεννήσεων για κάθε περιοχή. Έπειτα, ξεκινά η μέτρηση του χρόνου για την αξιολόγηση της απόδοσης του QuickSort.

Μέτρηση χρόνου: Η διάρκεια εκτέλεσης του QuickSort μετριέται με τη βοήθεια του <chrono>, και υπολογίζεται ο μέσος χρόνος εκτέλεσης από 1000 επαναλήψεις του αλγορίθμου.

Έξοδος: Τα αποτελέσματα της ταξινόμησης εκτυπώνονται με την PrintSummedCounts(), ενώ ο μέσος χρόνος εκτέλεσης του QuickSort εμφανίζεται στην έξοδο.

```
Summed Birth Counts for each region between 2005 and 2022 :
```

```
Region: "Region not stated or area outside region", Sum: 792
Region: West Coast region, Sum: 6747
Region: Tasman region, Sum: 8805
Region: Marlborough region, Sum: 9147
Region: Nelson region, Sum: 9861
Region: Gisborne region, Sum: 13041
Region: Southland region, Sum: 21960
Region: Taranaki region, Sum: 27327
Region: Hawke's Bay region, Sum: 38973
Region: Northland region, Sum: 40191
Region: Otago region, Sum: 40845
Region: Manawatu-Wanganui region, Sum: 55497
Region: Bay of Plenty region, Sum: 70983
Region: Waikato region, Sum: 109491
Region: Wellington region, Sum: 111969
Region: Canterbury region, Sum: 125601
Region: Auckland region, Sum: 391893
Average Quicksort execution time: 66 microseconds
```


Πειραματική συγκριση

Merge Sort:

Είναι ένας σταθερός αλγόριθμος ταξινόμησης με πολυπλοκότητα $O(n \log n)$ στην χειρότερη, μέση και καλύτερη περίπτωση.

Quick Sort:

Έχει πολυπλοκότητα $O(n \log n)$ στην μέση περίπτωση, αλλά μπορεί να φτάσει $O(n^2)$ στην χειρότερη περίπτωση λόγω κακής επιλογής του pivot.

Ωστόσο, είναι γενικά πιο γρήγορος από τον Merge Sort λόγω της μικρότερης σταθεράς και της in-place φύσης του, δηλαδή

δεν χρειάζεται επιπλέον χώρο μνήμης εκτός από το ίδιο το array.

Με βάση τον μέσο χρόνο εκτέλεσης του κάθε αλγόριθμου, παρατηρώ ότι ο MergeSort είναι πιο γρήγορος στην πλειονότητα των περιπτώσεων.

Είναι πιο γρήγορος από τον quicksort, διότι σε κάθε περίπτωση έχει σταθερό μέσο χρόνο εκτέλεσης ενώ ο Quick Sort δείχνει μεγαλύτερη διακύμανση, λόγω

του τρόπου που επιλέγονται τα pivot σε κάθε αναδρομική κλήση.

Heap Sort και Counting Sort

Για την υλοποίηση του δεύτερου ερωτήματος φτιάξαμε ένα c++ αρχείο με όνομα “[heapsort.cpp](#)”. Ακολουθήσαμε τα βήματα του αλγόριθμου και φτιάξαμε το παρακάτω πρόγραμμα.

```
#include "read_print.h" // Access read_print.h
#include <iostream>
#include <string>
#include <chrono>        // Include chrono for time measurements

using namespace std;
using namespace chrono;

// Heapify function to maintain heap property
void heapify(SummedCount arr[], int n, int i) {
    int largest = i; // Initialize largest as root
    int l = 2 * i + 1; // Calculate left child index
    int r = 2 * i + 2; // Calculate right child index

    // If left child is larger than root
    if (l < n && arr[l].Sum > arr[largest].Sum)
        largest = l; // Update largest to left child index

    // If right child is larger than largest so far
    if (r < n && arr[r].Sum > arr[largest].Sum)
        largest = r; // Update largest to right child index

    // If largest is not root
    if (largest != i) {
        swap(arr[i], arr[largest]); // Swap root with largest

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest); // Recursively call heapify for the affected
sub-tree
    }
}
```

```

// Heap sort algorithm
void heapSort(SummedCount arr[], int n) {
    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i); // Call heapify to build the heap

    // One by one extract an element from heap
    for (int i = n - 1; i > 0; i--) {
        // Move current root to end
        swap(arr[0], arr[i]); // Swap root with last element

        // Call max heapify on the reduced heap
        heapify(arr, i, 0); // Call heapify to maintain heap property
    }
}

int main() {
    Read_Data(); // Call Read_Data() to read the data

    CalculateDeathSums(); // Calculate and store summed death counts for each
region between 2005 and 2022

    // Start measuring time
    auto start_time = high_resolution_clock::now();

    const int iterations = 1000; // Number of iterations to find average
execution time of the algorithm
    for (int i = 0; i < iterations; ++i) {

        // Sort Summedcounts based on the total sum of each region in ascending
order using HeapSort
        heapSort(Summedcounts, MAXSUMS);
    }

    // Stop measuring time
    auto end_time = high_resolution_clock::now();

    // Calculate duration of the sorting
    auto duration = duration_cast<microseconds>(end_time - start_time); //
Calculate the duration in microseconds

    // Print the contents of the Summedcounts array
    cout << "Summed Death Counts for each region between 2005 and 2022 :" << endl
<< endl;
    PrintSummedCounts(Summedcounts, MAXSUMS);
}

```

```

    // Print the average execution time of heapsort
    cout << "Average HeapSort execution time: " << duration.count() / iterations
<< " microseconds" << endl;

    return 0;
}

```

Αυτός ο κώδικας υλοποιεί τον αλγόριθμο HeapSort για την ταξινόμηση του πίνακα δομών “SummedCount”, ο οποίος περιέχει στοιχεία που αναπαριστούν άθροισμα και μέτρηση θανάτων για κάθε περιοχή και έτος μεταξύ 2005 και 2022.

Βιβλιοθήκες και ονοματοχώροι: Ο κώδικας χρησιμοποιεί τις βιβλιοθήκες <iostream>, <string> για την επεξεργασία εισόδου/εξόδου και <chrono> για τη μέτρηση του χρόνου.

Δομή “SummedCount”: Περιέχει τουλάχιστον ένα πεδίο “Sum”, το οποίο χρησιμοποιείται για τη σύγκριση και την ταξινόμηση των δεδομένων.

heapify(): Η συνάρτηση heapify() χρησιμοποιείται για να διατηρήσει την ιδιότητα του σωρού (heap property) ενός υποδέντρου, όπου το κύριο στοιχείο (root) είναι μεγαλύτερο από τα παιδιά του.

heapSort(): Η κύρια συνάρτηση heapSort() καλείται για να ταξινομήσει τον πίνακα SummedCount με χρήση του αλγορίθμου HeapSort. Αρχικά, καλεί τη heapify() για να δημιουργήσει έναν σωρό (heap) από τον πίνακα, και στη συνέχεια επαναλαμβάνει την διαδικασία για να ταξινομήσει τον πίνακα σε αύξουσα σειρά.

main(): Στη συνάρτηση main(), γίνεται κλήση της Read_Data() για την ανάγνωση δεδομένων και της CalculateDeathSums() για τον υπολογισμό των συνολικών αθροισμάτων θανάτων για κάθε περιοχή. Έπειτα, ξεκινά η μέτρηση του χρόνου για την αξιολόγηση της απόδοσης του HeapSort.

Μέτρηση χρόνου: Η διάρκεια εκτέλεσης του HeapSort μετρείται με τη βοήθεια του <chrono>, και υπολογίζεται ο μέσος χρόνος εκτέλεσης από 1000 επαναλήψεις του αλγορίθμου.

Έξοδος: Τα αποτελέσματα της ταξινόμησης εκτυπώνονται με την PrintSummedCounts(), ενώ ο μέσος χρόνος εκτέλεσης του HeapSort εμφανίζεται στην έξοδο.

```
Summed Death Counts for each region between 2005 and 2022 :  
  
Region: "Region not stated or area outside region", Sum: 939  
Region: West Coast region, Sum: 5070  
Region: Tasman region, Sum: 6501  
Region: Gisborne region, Sum: 7140  
Region: Marlborough region, Sum: 7218  
Region: Nelson region, Sum: 7560  
Region: Southland region, Sum: 14745  
Region: Taranaki region, Sum: 17424  
Region: Hawke's Bay region, Sum: 24861  
Region: Northland region, Sum: 26223  
Region: Otago region, Sum: 28950  
Region: Manawatu-Wanganui region, Sum: 36753  
Region: Bay of Plenty region, Sum: 43290  
Region: Waikato region, Sum: 55626  
Region: Wellington region, Sum: 57162  
Region: Canterbury region, Sum: 78264  
Region: Auckland region, Sum: 143199  
Average HeapSort execution time: 32 microseconds
```

Για την υλοποίηση της Counting Sort φτιαξαμε ενα c++ αρχειο με όνομα “[CountingSort.cpp](#)”. Ακολουθήσαμε τα βήματα του αλγόριθμου και φτιάξαμε το παρακάτω πρόγραμμα.

```
#include "read_print.h" // Access read_print.h  
#include <iostream>  
#include <string>  
#include <chrono> // Include chrono for time measurements  
  
using namespace std;  
using namespace chrono;  
  
// Function to calculate the maximum value in Summedcounts array  
int getMax(SummedCount arr[], int n) {  
    int max = arr[0].Sum;  
    for (int i = 1; i < n; i++) {  
        if (arr[i].Sum > max) {  
            max = arr[i].Sum;  
        }  
    }  
}
```

```

        return max;
    }

    // Counting Sort function
    void countingSort(SummedCount arr[], int n) {
        // Find the maximum element in the array
        int max = getMax(arr, n);
        int* count = new int[max + 1] {0}; // Create count array and initialize to 0
        SummedCount* output = new SummedCount[n]; // Create output array

        // Store the count of each element
        for (int i = 0; i < n; i++) {
            count[arr[i].Sum]++;
        }

        // Change count[i] so that count[i] now contains the actual position of this
        // element in output array
        for (int i = 1; i <= max; i++) {
            count[i] += count[i - 1];
        }

        // Build the output array
        for (int i = n - 1; i >= 0; i--) {
            output[count[arr[i].Sum] - 1] = arr[i];
            count[arr[i].Sum]--;
        }

        // Copy the output array to arr, so that arr now contains sorted elements
        for (int i = 0; i < n; i++) {
            arr[i] = output[i];
        }

        delete[] count;
        delete[] output;
    }

    int main() {
        Read_Data(); // Call Read_Data() to read the data

        CalculateDeathSums(); // Calculate and store summed death counts for each
        // region between 2005 and 2022

        // Start measuring time
        auto start_time = high_resolution_clock::now();
    }

```

```

    const int iterations = 1000; // Number of iterations to find average
    execution time of the algorithm
    for (int i = 0; i < iterations; ++i) {

        // Sort Summedcounts based on the total sum of each region in ascending
        order using Counting Sort
        countingSort(Summedcounts, MAXSUMS);
    }

    // Stop measuring time
    auto end_time = high_resolution_clock::now();

    // Calculate duration of the sorting
    auto duration = duration_cast<microseconds>(end_time - start_time); //
    Calculate the duration in microseconds

    // Print the contents of Summedcounts array
    cout << "Summed Death Counts for each region between 2005 and 2022 : " << endl
    << endl;
    PrintSummedCounts(Summedcounts, MAXSUMS);

    // Print the average execution time of Counting sort
    cout << "Average Counting Sort execution time: " << duration.count() /
    iterations << " microseconds" << endl;

    return 0;
}

```

Αυτός ο κώδικας υλοποιεί τον αλγόριθμο Counting Sort για την ταξινόμηση του πίνακα δομών “SummedCount”, ο οποίος περιέχει στοιχεία που αναπαριστούν άθροισμα θανάτων για κάθε περιοχή και έτος μεταξύ 2005 και 2022.

Βιβλιοθήκες και ονοματοχώροι: Ο κώδικας χρησιμοποιεί τις βιβλιοθήκες <iostream>, <string> για την επεξεργασία εισόδου/εξόδου και <chrono> για τη μέτρηση του χρόνου.

Δομή “SummedCount”: Περιέχει τουλάχιστον ένα πεδίο “Sum”, το οποίο χρησιμοποιείται για τη σύγκριση και την ταξινόμηση των δεδομένων.

getMax(): Η συνάρτηση getMax() χρησιμοποιείται για να εντοπίσει τη μέγιστη τιμή του πεδίου “Sum” στον πίνακα “SummedCount”.

countingSort(): Η κύρια συνάρτηση countingSort() υλοποιεί τον αλγόριθμο Counting Sort. Αρχικά, υπολογίζει τη συχνότητα κάθε τιμής “Sum” και την αποθηκεύει στον πίνακα “count”. Στη συνέχεια, χρησιμοποιεί τον πίνακα “count” για να οργανώσει τα στοιχεία του “arr” στον πίνακα “output” με τη σειρά, βασιζόμενη στην τιμή του “Sum”.

main(): Στη συνάρτηση main(), γίνεται κλήση της Read_Data() για την ανάγνωση δεδομένων και της CalculateDeathSums() για τον υπολογισμό των συνολικών αθροισμάτων θανάτων για κάθε περιοχή. Έπειτα, ξεκινά η μέτρηση του χρόνου για την αξιολόγηση της απόδοσης του Counting Sort.

Μέτρηση χρόνου: Η διάρκεια εκτέλεσης του Counting Sort μετρείται με τη βοήθεια του <chrono>, και υπολογίζεται ο μέσος χρόνος εκτέλεσης από 1000 επαναλήψεις του αλγορίθμου.

Έξοδος: Τα αποτελέσματα της ταξινόμησης εκτυπώνονται με την PrintSummedCounts(), ενώ ο μέσος χρόνος εκτέλεσης του Counting Sort εμφανίζεται στην έξοδο.

```
Summed Death Counts for each region between 2005 and 2022 :
```

```
Region: "Region not stated or area outside region", Sum: 939
```

```
Region: West Coast region, Sum: 5070
```

```
Region: Tasman region, Sum: 6501
```

```
Region: Gisborne region, Sum: 7140
```

```
Region: Marlborough region, Sum: 7218
```

```
Region: Nelson region, Sum: 7560
```

```
Region: Southland region, Sum: 14745
```

```
Region: Taranaki region, Sum: 17424
```

```
Region: Hawke's Bay region, Sum: 24861
```

```
Region: Northland region, Sum: 26223
```

```
Region: Otago region, Sum: 28950
```

```
Region: Manawatu-Wanganui region, Sum: 36753
```

```
Region: Bay of Plenty region, Sum: 43290
```

```
Region: Waikato region, Sum: 55626
```

```
Region: Wellington region, Sum: 57162
```

```
Region: Canterbury region, Sum: 78264
```

```
Region: Auckland region, Sum: 143199
```

```
Average Counting Sort execution time: 904 microseconds
```


Πειραματική σύγκριση

Heap Sort:

Είναι ένας αλγόριθμος ταξινόμησης με πολυπλοκότητα $O(n \log n)$ σε όλες τις περιπτώσεις.

Είναι in-place, δηλαδή δεν χρειάζεται επιπλέον χώρο μνήμης εκτός από το ίδιο το array.

Είναι σχετικά αργός σε σχέση με άλλους αλγορίθμους όπως ο Quick Sort ή ο Merge Sort λόγω της φύσης της διαχείρισης του σωρού.

Counting Sort:

Είναι ένας αλγόριθμος ταξινόμησης με πολυπλοκότητα $O(n + k)$, όπου n είναι ο αριθμός των στοιχείων και k είναι το εύρος των τιμών.

Είναι ιδιαίτερα αποδοτικός όταν το εύρος των τιμών (k) είναι μικρό σε σύγκριση με το πλήθος των στοιχείων (n).

Δεν είναι in-place, καθώς απαιτεί επιπλέον χώρο για τους πίνακες καταμέτρησης και εξόδου.

Μπορεί να είναι πολύ αποδοτικός όταν τα δεδομένα είναι περιορισμένα σε εύρος τιμών, αλλά μπορεί να καταστεί αναποτελεσματικός όταν το εύρος είναι μεγάλο.

Με βάση τον μέσο χρόνο εκτέλεσης του κάθε αλγόριθμου, παρατηρώ ότι ο heap Sort είναι πιο γρήγορος στην πλειονότητα των περιπτώσεων.

Αυτό οφείλεται στο γεγονός ότι ο Counting Sort έχει χειρότερη πολυπλοκότητα χρόνου στην περίπτωση όπου το

εύρος των τιμών που πρέπει να ταξινομηθούν είναι σημαντικά μεγαλύτερο από τον αριθμό των στοιχείων.

Διαδική Αναζήτηση και Αναζήτηση Παρεμβολής

Για την υλοποίηση του τρίτου ερωτήματος φτιάξαμε ένα c++ αρχείο με όνομα “BinarySearch.cpp”. Ακολουθήσαμε τα βήματα του αλγόριθμου και φτιάξαμε το παρακάτω πρόγραμμα.

```
#include "heapsort.h" //access heapsort.h
#include <iostream>
#include <string>
#include <chrono> // Include chrono for time measurements

using namespace std;
using namespace chrono;

bool printOnce= false; //boolean value to print the found counts once

void BinarySearch(SummedCount Summedcounts[], int size, int b1, int b2) {

    int left = 0, right = size - 1; // define the index left and right of the
array
    int leftBoundary = -1;           // leftboundary = the first sum to be
greater than b1

    while (left <= right) {          //check if all sums between b1 and b2 have been
searched
        int mid = left + (right - left) / 2; //Calculate median
        int i =0;

        if (Summedcounts[mid].Sum >= b1 && Summedcounts[i].Sum <= b2) { // if
the sum of the median index is >= b1 and the sum of the i index is <= b2
                                                                    //
then make the median the leftboundary
            leftBoundary = mid;
            right = mid - 1; //update the number of right to the updated
number of cells in Summedcounts
            i += mid; //update i by moving it mid number positions
right
        } else {
            left = mid + 1; //if the sum of the median index is <= b1 and
the sum of the i index >=b2, update left by moving it one cell after the median
```

```

    }
}

if (!printOnce){ //if printOnce is false then print the found regions

    // If no valid boundary found, return
    if (leftBoundary == -1) {
        cout << "No regions found with summed birth counts in the given range."
<< endl;
        return;
    }

    // Print regions with summed birth counts in the range [b1, b2]
    cout << "Regions with summed birth counts in the range [" << b1 << ", " << b2
<< "]" << endl;

    for (int i = leftBoundary; i < size && Summedcounts[i].Sum <= b2; ++i) {
        cout << "Region: " << Summedcounts[i].Region << ", Sum: " <<
Summedcounts[i].Sum << endl;
    }
}
}

int main()
{

    Read_Data(); //access Read_Data() from read_print.h

    CalculateBirthSums(); // Calculate and store summed birth counts for each
region between 2005 and 2022

    // Sort the Summedcounts array using heap sort based on the Sum value
    heapSort(Summedcounts, MAXSUMS);

    int b1, b2; //define bound b1 and bound b2
    cout << "Enter the lower bound (b1) and upper bound (b2) for the birth counts
range: ";
    cin >> b1 >> b2;

    // Start measuring time
    auto start_time = high_resolution_clock::now();

```

```

    const int iterations = 2000; // Number of iterations to find average
    execution time of the algorithm
    for (int i = 0; i < iterations; ++i) {

        BinarySearch(Summedcounts, MAXSUMS, b1, b2); // Perform Binary Search on the
data
        printOnce = true;
    }
    // Stop measuring time
    auto end_time = high_resolution_clock::now();

    auto duration = duration_cast<microseconds>(end_time - start_time); //
Calculate the duration in microseconds

    // Print the average execution time of Binary Search
    cout << "Binary Search execution time: " << duration.count() / iterations <<
" microseconds" << endl;

    return 0;
}

```

Αυτός ο κώδικας περιέχει ένα πρόγραμμα C++ που περιλαμβάνει την υλοποίηση ενός δυαδικού αλγορίθμου αναζήτησης (Binary Search) για την αναζήτηση περιοχών στις οποίες η συνολική γέννηση είναι στο εύρος [b1, b2].

Βιβλιοθήκες και Ορισμοί:

Χρησιμοποιούνται οι βιβλιοθήκες <iostream>, <string> και <chrono> για τις βασικές εισόδους/εξόδους, τη διαχείριση συμβολοσειρών και τη μέτρηση του χρόνου αντίστοιχα.

Η “SummedCount” είναι η δομή δεδομένων που περιέχει πληροφορίες περιοχών (Region) και συνολικής γέννησης (Sum).

Δυαδική Αναζήτηση (BinarySearch):

Αναζητεί πρώτα την αριστερή συνριοθέτηση (leftBoundary) των περιοχών όπου η συνολική γέννηση είναι τουλάχιστον b1.

Χρησιμοποιεί δυαδική αναζήτηση σε ταξινομημένο πίνακα (Summedcounts[]) για αποδοτική αναζήτηση.

Αφού βρει την leftBoundary, εκτυπώνει τις περιοχές όπου η συνολική γέννηση είναι στο εύρος [b1, b2].

Κύρια Συνάρτηση (main):

Καλεί δύο σημαντικές λειτουργίες Read_Data() και CalculateBirthSums() που δεν παρατίθενται εδώ, αλλά προφανώς διαβάζουν δεδομένα και υπολογίζουν τις συνολικές γεννήσεις για κάθε περιοχή από το 2005 έως το 2022.

Ταξινομεί τον πίνακα Summedcounts χρησιμοποιώντας heap sort με βάση την τιμή “Sum”.

Ζητά από το χρήστη να εισάγει τα όρια b1 και b2.

Μετρά και εκτυπώνει τον μέσο χρόνο εκτέλεσης της δυαδικής αναζήτησης για 2000 επαναλήψεις, χρησιμοποιώντας την <chrono> για μέτρηση του χρόνου.

Παράδειγμα εκτέλεσης του προγράμματος:

```
Enter the lower bound (b1) and upper bound (b2) for the birth counts range: 1 10000000
Regions with summed birth counts in the range [1, 10000000]:
Region: "Region not stated or area outside region", Sum: 792
Region: West Coast region, Sum: 6747
Region: Tasman region, Sum: 8805
Region: Marlborough region, Sum: 9147
Region: Nelson region, Sum: 9861
Region: Gisborne region, Sum: 13041
Region: Southland region, Sum: 21960
Region: Taranaki region, Sum: 27327
Region: Hawke's Bay region, Sum: 38973
Region: Northland region, Sum: 40191
Region: Otago region, Sum: 40845
Region: Manawatu-Wanganui region, Sum: 55497
Region: Bay of Plenty region, Sum: 70983
Region: Waikato region, Sum: 109491
Region: Wellington region, Sum: 111969
Region: Canterbury region, Sum: 125601
Region: Auckland region, Sum: 391893
Binary Search execution time: 16 microseconds
```

Για την υλοποίηση της Interpolation Search φτιάξαμε ένα c++ αρχείο με όνομα “[InterpolationSearch.cpp](#)”. Ακολουθήσαμε τα βήματα του αλγόριθμου και φτιάξαμε το παρακάτω πρόγραμμα.

```

#include "heapsort.h" //access heapsort.h
#include <iostream>
#include <string>
#include <chrono> // Include chrono for time measurements

using namespace std;
using namespace chrono;

bool printOnce= false; //boolean value to print the found counts once

void InterpolationSearch(SummedCount Summedcounts[], int size, int b1, int b2) {

    int low = 0, high = size - 1; // define the index low and high of the array
    int leftBoundary = -1; // leftboundary = the first sum to be greater than b1
    bool countsFound = false; // Flag to track if counts are found within the
range

    // Perform Interpolation Search to find the left boundary (first element >=
b1)
    while (low <= high && b1 >= Summedcounts[low].Sum && b1 <=
Summedcounts[high].Sum) {
        if (low == high) { // If low and high point to the same element
            if (Summedcounts[low].Sum == b1)
                leftBoundary = low; // Set leftBoundary if element equals b1
            break;
        }

        // Calculate the position using the interpolation formula
        int pos = low + (((high - low) / (Summedcounts[high].Sum -
Summedcounts[low].Sum)) * (b1 - Summedcounts[low].Sum));

        if (Summedcounts[pos].Sum == b1) {
            leftBoundary = pos; // If element at pos equals b1, leftBoundary =
pos
            break;
        }

        if (Summedcounts[pos].Sum < b1) { // If element at pos less than b1
            low = pos + 1; //update the number of index low to the updated number
of cells in Summedcounts
        }
        // If element at pos greater than b1
    } else {

```

```

        high = pos - 1; //update the number of index high to the updated
number of cells in Summedcounts
    }
}

if (!printOnce){ //if printOnce is false then print the found regions
if (leftBoundary == -1) {
    leftBoundary = low; // If leftBoundary not found, set it to low
}

    cout << "Regions with summed birth counts in the range [" << b1 << ", " << b2
<< "]: " << endl;

    // Iterate from leftBoundary to the end or until the sum exceeds b2
    for (int i = leftBoundary; i < size && Summedcounts[i].Sum <= b2; ++i) {
        cout << "Region: " << Summedcounts[i].Region << ", Sum: " <<
Summedcounts[i].Sum << endl;
        countsFound = true; // Set flag to true if any counts are found within
the range
    }

    if (!countsFound) { // if countsfound = false
        cout << "No regions found with summed birth counts in the given range."
<< endl;
    }
}
}

int main() {
    Read_Data(); //access Read_Data() from read_print.h
    CalculateBirthSums(); // Calculate and store summed birth counts for each
region between 2005 and 2022

    // Sort the Summedcounts array using heap sort based on the Sum value
    heapSort(Summedcounts, MAXSUMS);

    int b1, b2; //define bound b1 and bound b2
    cout << "Enter the lower bound (b1) and upper bound (b2) for the birth counts
range: ";
    cin >> b1 >> b2;

    auto start_time = high_resolution_clock::now(); // Start measuring time

```

```

    const int iterations = 2000; // Number of iterations to find average
    execution time of the algorithm
    for (int i = 0; i < iterations; ++i) {

        InterpolationSearch(Summedcounts, MAXSUMS, b1, b2); // Perform interpolation
        search on the data
        printOnce = true;
    }

    auto end_time = high_resolution_clock::now(); // Stop measuring time

    auto duration = duration_cast<microseconds>(end_time - start_time); //
    Calculate the duration in microseconds
    // Print the average execution time of Interpolation Search
    cout << "Interpolation Search execution time: " << duration.count()
    /iterations << " microseconds" << endl;

    return 0;
}

```

Αυτός ο κώδικας περιλαμβάνει ένα πρόγραμμα C++ που χρησιμοποιεί τη μέθοδο αναζήτησης με Παρεμβολή (Interpolation Search) για να βρει περιοχές όπου η συνολική γέννηση είναι εντός ενός καθορισμένου εύρους [b1, b2].

Βιβλιοθήκες και Δήλωσεις:

Περιλαμβάνονται οι βιβλιοθήκες <iostream>, <string> και <chrono> για τις βασικές εισόδους/εξόδους, τη διαχείριση συμβολοσειρών και τη μέτρηση του χρόνου αντίστοιχα.

Υπάρχει μια δήλωση printOnce που χρησιμοποιείται για να ελέγχει αν έχουν εκτυπωθεί οι εύρεσης για το εύρος [b1, b2].

Συνάρτηση Αναζήτησης (InterpolationSearch):

Αρχικοποιείται η low και η high για τα όρια του πίνακα "Summedcounts".

Χρησιμοποιείται η μέθοδος της διασταύρωσης για να βρει τον leftBoundary, που είναι το πρώτο στοιχείο του πίνακα με συνολική γέννηση $\geq b1$.

Αν το leftBoundary δεν βρεθεί κατά τη διάρκεια της αναζήτησης, τότε ορίζεται στο low.

Εκτυπώνονται οι περιοχές όπου η συνολική γέννηση είναι στο εύρος [b1, b2], χρησιμοποιώντας το leftBoundary ως σημείο έναρξης.

Κύρια Συνάρτηση (main):

Καλείται η Read_Data() και η CalculateBirthSums() για την ανάγνωση δεδομένων και τον υπολογισμό των συνολικών γεννήσεων για κάθε περιοχή.

Ο ταξινομημένος πίνακας “Summedcounts” χρησιμοποιείται για την εφαρμογή του αλγορίθμου heapsort.

Ζητούνται από τον χρήστη τα όρια b1 και b2.

Η διαδικασία αναζήτησης με διασταύρωση (InterpolationSearch) εκτελείται επαναληπτικά για 2000 επαναλήψεις για τη μέτρηση του μέσου χρόνου εκτέλεσης.

Εκτυπώνεται ο μέσος χρόνος εκτέλεσης της αναζήτησης.

Παράδειγμα εκτέλεσης του προγράμματος:

```
Enter the lower bound (b1) and upper bound (b2) for the birth counts range: 1 10000000
Regions with summed birth counts in the range [1, 10000000]:
Region: "Region not stated or area outside region", Sum: 792
Region: West Coast region, Sum: 6747
Region: Tasman region, Sum: 8805
Region: Marlborough region, Sum: 9147
Region: Nelson region, Sum: 9861
Region: Gisborne region, Sum: 13041
Region: Southland region, Sum: 21960
Region: Taranaki region, Sum: 27327
Region: Hawke's Bay region, Sum: 38973
Region: Northland region, Sum: 40191
Region: Otago region, Sum: 40845
Region: Manawatu-Wanganui region, Sum: 55497
Region: Bay of Plenty region, Sum: 70983
Region: Waikato region, Sum: 109491
Region: Wellington region, Sum: 111969
Region: Canterbury region, Sum: 125601
Region: Auckland region, Sum: 391893
Interpolation Search execution time: 15 microseconds
```

Πειραματική σύγκριση

Δυαδική Αναζήτηση:

Ο αλγόριθμος αυτός απαιτεί ταξινομημένα δεδομένα.

Ο μέσος χρόνος περίπτωσης της δυαδικής αναζήτησης είναι $O(\log n)$, όπου n είναι το πλήθος των στοιχείων.

Επηρεάζεται από την κατανομή των δεδομένων, καθώς αν τα δεδομένα είναι ευαίσθητα στην ταξινόμηση,

(π.χ., εντοπίζονται σε συγκεκριμένες περιοχές), τότε μπορεί να επιτύχει πολύ καλή απόδοση.

Ωστόσο, εάν τα δεδομένα δεν είναι ομοιόμορφα κατανεμημένα ή δεν είναι καλά ταξινομημένα, ο αλγόριθμος μπορεί να αποδειχθεί λιγότερο αποδοτικός.

Αναζήτηση με Παρεμβολή:

Ο αλγόριθμος αυτός είναι μια εκτελεστική βελτίωση της δυαδικής αναζήτησης που προσπαθεί να εκτιμήσει περισσότερο τη θέση του στοιχείου που αναζητούμε.

Ο μέσος χρόνος περίπτωσης της αναζήτησης με παρεμβολή είναι $O(\log \log n)$ και ο χρόνος χειρότερης περίπτωσης είναι $O(n)$, όπου n είναι το πλήθος των στοιχείων.

Ο αλγόριθμος είναι επίσης ευαίσθητος στην κατανομή των δεδομένων και μπορεί να παρουσιάσει καλή απόδοση όταν τα δεδομένα είναι καλά διανεμημένα.

Ωστόσο, εάν τα δεδομένα δεν είναι ομοιόμορφα κατανεμημένα ή δεν ακολουθούν ένα μοντέλο πρόβλεψης, η απόδοση του αλγορίθμου αναζήτησης με διαδοχικές προσεγγίσεις μπορεί να είναι αντίστοιχα χειρότερη.

Με βάση τους μέσους χρόνους περίπτωσης του κάθε αλγορίθμου παρατηρούμε ότι ο αλγόριθμος Interpolation Search με μέσο χρόνο περίπτωσης $O(\log \log n)$ είναι σχετικά πιο αποδοτικός σε σχέση με τον αλγόριθμο Binary Search με μέσο χρόνο περίπτωσης $O(\log n)$.

Διαδική Αναζήτηση Παρεμβολής

Για την υλοποίηση του τέταρτου ερωτήματος φτιάξαμε ένα c++ αρχείο με όνομα “BinaryInterpolationSearch.cpp”. Ακολουθήσαμε τα βήματα του αλγορίθμου και φτιάξαμε το παρακάτω πρόγραμμα.

```
#include "heapsort.h" // access heapsort.h
#include <iostream>
#include <string>
#include <chrono>
#include <cmath> // Include cmath for sqrt

using namespace std;
using namespace chrono;

bool printOnce = false; //boolean value to print the found counts once

void BinaryInterpolationSearch(SummedCount Summedcounts[], int size, int b1, int b2) {
    int low = 0, high = size - 1; // define the index low and high of the array
    int leftBoundary = -1; // leftboundary = the first sum to be greater than b1
    bool countsFound = false; // Flag to track if counts are found within the range

    // Perform Interpolation Search to find the left boundary (first element >= b1)
    while (low <= high && b1 >= Summedcounts[low].Sum && b1 <= Summedcounts[high].Sum) {
        if (low == high) { // If low and high point to the same element
            if (Summedcounts[low].Sum == b1)
                leftBoundary = low; // Set leftBoundary if element equals b1
            break;
        }

        // Calculate the position using the interpolation formula
        int pos = low + (((high - low) / (Summedcounts[high].Sum - Summedcounts[low].Sum)) * (b1 - Summedcounts[low].Sum));

        if (Summedcounts[pos].Sum == b1) { // If element at pos equals b1,
            leftBoundary = pos;
        }
    }
}
```

```

        break;
    }

    if (Summedcounts[pos].Sum < b1) { // If element at pos less than b1
        low = pos + 1; //update the number of index low to the updated
number of cells in Summedcounts

        // If element at pos greater than b1
    } else {
        high = pos - 1; //update the number of index high to the updated
number of cells in Summedcounts
    }

    int size = high - low + 1; //update size

    if (size <= 3) {
        int next = (size* (b1 - Summedcounts[low].Sum) /
(Summedcounts[high].Sum - Summedcounts[low].Sum)); //if updated size is <=3
calculate next

        while (Summedcounts[next].Sum != b1) { //while sumcount at next
index is not b1
            int i = 0; // define i=0
            size = high - low + 1; //update size

            if ( Summedcounts[next].Sum < b1) { //if sumcount at index
next < b1
                while (b1 > Summedcounts[next + i * (int)sqrt(size) -
1].Sum) {
                    i++; //increment i while b1> summedcounts at
index next + i*root(size) -1
                }
                high = next + i * sqrt(size); //update high to next + i
* root(size)
            } else {
                while (b1 < Summedcounts[next - i * (int)sqrt(size) +
1].Sum) {
                    i++; //increment i while b1< summedcounts at
index next - i*root(size) + 1
                }
                high = next - (i - 1) * sqrt(size); //update high to
next - (i-1)*root(size)
            }
            low = next - i * sqrt(size); //update low to next - i*
root(size)

```

```

        next = low + ((high - low + 1) * (b1 - Summedcounts[low].Sum)) /
(Summedcounts[high].Sum - Summedcounts[low].Sum); //update next
        if (Summedcounts[next].Sum == b1) { //if sumcount at index
next = b1
            leftBoundary = next; //leftboundary found at
next
                break;
            } else {
                return; //else return
            }
        }
    }

    if (!printOnce) { //if printOnce is false then print the found regions
        if (leftBoundary == -1) {
            leftBoundary = low; // If leftBoundary not found, set it to low
        }

        cout << "Regions with summed birth counts in the range [" << b1 << ", "
<< b2 << "]:" << endl;

        // Iterate from leftBoundary to the end or until the sum exceeds b2
        for (int i = leftBoundary; i < size && Summedcounts[i].Sum <= b2; ++i) {
            cout << "Region: " << Summedcounts[i].Region << ", Sum: " <<
Summedcounts[i].Sum << endl;
            countsFound = true; // Set flag to true if any counts are found
within the range
        }

        if (!countsFound) { // if countsfound = false
            cout << "No regions found with summed birth counts in the given
range." << endl;
        }
    }
}

int main() {
    Read_Data(); //access Read_Data() from read_print.h
    CalculateBirthSums(); // Calculate and store summed birth counts for each
region between 2005 and 2022

    // Sort the Summedcounts array using heap sort based on the Sum value
    heapSort(Summedcounts, MAXSUMS);
}

```

```

    int b1, b2; //define bound b1 and bound b2
    cout << "Enter the lower bound (b1) and upper bound (b2) for the birth counts
range: ";
    cin >> b1 >> b2;

    auto start_time = high_resolution_clock::now(); // Start measuring time

    const int iterations = 2000; // Number of iterations to find average
execution time of the algorithm
    for (int i = 0; i < iterations; ++i) {

        BinaryInterpolationSearch(Summedcounts, MAXSUMS, b1, b2); // Perform Binary
interpolation search on the data
        printOnce = true;
    }

    auto end_time = high_resolution_clock::now(); // Stop measuring time

    auto duration = duration_cast<microseconds>(end_time - start_time); //
Calculate the duration in microseconds
    // Print the average execution time of Interpolation Search
    cout << "Binary Interpolation Search execution time: " << duration.count()
/iterations << " microseconds" << endl;

    return 0;
}

```

Αυτός ο κώδικας υλοποιεί τη μέθοδο αναζήτησης με διασταύρωση (Binary Interpolation Search) για την εύρεση περιοχών με συνολικές γεννήσεις σε ένα καθορισμένο εύρος [b1, b2].

Βιβλιοθήκες και Δήλωσεις:

Περιλαμβάνονται οι βιβλιοθήκες <iostream>, <string>, <chrono> και <cmath> για τη διαχείριση εισόδου/εξόδου, την χρονομέτρηση και τις μαθηματικές λειτουργίες όπως η τετραγωνική ρίζα (sqrt).

Ορίζεται η μεταβλητή printOnce για να ελέγχει αν έχουν εκτυπωθεί οι ευρεσιτεχνίες για το εύρος [b1, b2].

Συνάρτηση BinaryInterpolationSearch:

Αρχικοποιούνται οι μεταβλητές low και high ως τα όρια του πίνακα Summedcounts.

Χρησιμοποιείται η μέθοδος αναζήτησης με διασταύρωση για να βρει το leftBoundary, που είναι το πρώτο στοιχείο με συνολική γέννηση $\geq b1$.

Υπάρχει έλεγχος μέγεθους του παραθύρου αναζήτησης. Όταν το μέγεθος γίνεται μικρότερο ή ίσο του 3, η αναζήτηση μεταβαίνει σε γραμμική αναζήτηση.

Εκτυπώνονται οι περιοχές όπου η συνολική γέννηση είναι στο εύρος [b1, b2] από το leftBoundary έως το τέλος του πίνακα ή όταν η συνολική γέννηση υπερβαίνει το b2.

Υπάρχει επίσης έλεγχος για τυχόν περιοχές που δεν βρέθηκαν εντός του εύρους [b1, b2].

Κύρια Συνάρτηση (main):

Καλούνται οι συναρτήσεις Read_Data() και CalculateBirthSums() για την ανάγνωση δεδομένων και τον υπολογισμό των συνολικών γεννήσεων για κάθε περιοχή.

Ο ταξινομημένος πίνακας Summedcounts χρησιμοποιείται για την εφαρμογή του αλγορίθμου heapsort.

Ζητούνται από τον χρήστη τα όρια b1 και b2.

Η αναζήτηση BinaryInterpolationSearch εκτελείται επαναληπτικά για 2000 επαναλήψεις για τη μέτρηση του μέσου χρόνου εκτέλεσης.

Εκτυπώνεται ο μέσος χρόνος εκτέλεσης της αναζήτησης.

```
Enter the lower bound (b1) and upper bound (b2) for the birth counts range: 1 10000000
Regions with summed birth counts in the range [1, 10000000]:
Region: "Region not stated or area outside region", Sum: 792
Region: West Coast region, Sum: 6747
Region: Tasman region, Sum: 8805
Region: Marlborough region, Sum: 9147
Region: Nelson region, Sum: 9861
Region: Gisborne region, Sum: 13041
Region: Southland region, Sum: 21960
Region: Taranaki region, Sum: 27327
Region: Hawke's Bay region, Sum: 38973
Region: Northland region, Sum: 40191
Region: Otago region, Sum: 40845
Region: Manawatu-Wanganui region, Sum: 55497
Region: Bay of Plenty region, Sum: 70983
Region: Waikato region, Sum: 109491
Region: Wellington region, Sum: 111969
Region: Canterbury region, Sum: 125601
Region: Auckland region, Sum: 391893
Binary Interpolation Search execution time: 13 microseconds
```

Πειραματική σύγκριση

Ο μέσος χρόνος περίπτωσης της δυαδικής αναζήτησης με παρεμβολή είναι $O(\log \log n)$ και ο χρόνος χειρότερης περίπτωσης στην δική μας υλοποίηση είναι $O(n)$ και όχι $O(\sqrt{n})$ διότι στην υλοποίηση του αλγόριθμου έχουμε χρησιμοποιήσει loop μέσα σε loop, όπου n είναι το πλήθος των στοιχείων.

Η χειρότερη περίπτωση πολυπλοκότητας του αλγορίθμου Binary Interpolation Search μπορεί να αναλυθεί λαμβάνοντας υπόψη σενάρια όπου η αναζήτηση πρέπει να διασχίσει ένα σημαντικό μέρος του πίνακα πολλές φορές.

Στη χειρότερη περίπτωση, ο αλγόριθμος BIS μπορεί να απαιτήσει περισσότερα βήματα για να βρει τον στόχο σε σύγκριση με τον Βελτιωμένο αλγόριθμο BIS. Αυτό συμβαίνει επειδή ο δεύτερος αλγόριθμος επιτρέπει μεγαλύτερα βήματα κατά την αναζήτηση λόγω της εκθετικής αύξησης, οδηγώντας σε ταχύτερη σύγκλιση προς τον στόχο.

Επομένως στην πλειονότητα των εκτελέσεων του βελτιωμένου BIS αλγόριθμου παρατηρούμε ότι είναι γρηγορότερος από τον BIS χωρίς την παραλλαγή.

A. Δυαδικό Δέντρο Αναζήτησης

Ακολουθήσαμε παρόμοια διαδικασία ανάπτυξης όπως και στο πρώτο μέρος με τη διαφορά ότι φτιάξαμε δεύτερη δομή Node για τα υποδέντρα και προσθέσαμε λειτουργίες πάνω στο δυαδικό δέντρο αναζήτησης που θα καλούνται μέσα από το μενού επιλογών του χρήστη. Το δυαδικό δέντρο αποτελείται από κόμβους, όπου ο καθένας αποτελείται από διαφορετική εγγραφή και διατάσσονται όλοι με βάση την περιοχή.

```
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
using namespace std;

struct Row {
    int Period;
    bool Birth_Death;    // 0=Death , 1=Birth
    string Region;
    int Count;
};

// Structure of a BST node
struct Node {
    Row data;
    Node* left; //pointer to left subtree
    Node* right; //pointer to right subtree
};

// Function to create a new node that returns itself
Node* createNode(Row data) {
    Node* newNode = new Node();    //make newNode that points to struct Node
    newNode->data = data;           // copy elements of struct row and transfer
    them to struct newNode
    newNode->left = newNode->right = nullptr;    //make left and right subtree
    pointer null
    return newNode;                //return new node
}

// Function to insert a new node into the BST
void insertNode(Node*& root, Row data) {

    // Only insert if the data is for births
```

```

    if (!data.Birth_Death) {
        return;
    }

    if (root == nullptr) {          //if there is no root create one
        root = createNode(data);
        return;
    }

    // Compare the new region with the current node's region to decide where to
insert
    if (data.Region < root->data.Region) {    // if the region I want to insert is
alphabetically < than the root's
        insertNode(root->left, data);        // then insert it as a left child of
the root

    } else if (data.Region > root->data.Region) { // if the region I want to
insert is alphabetically > than the root's
        insertNode(root->right, data);      // then insert it as a right
child of the root

    } else {
        // If region is the same, decide based on period
        if (data.Period < root->data.Period) { // if the period of the region I
want to insert is < than the root's
            insertNode(root->left, data);    // then insert it as a left child
of the root

        } else {                          // if the period of the region I
want to insert is > than the root's
            insertNode(root->right, data);   // then insert it as a right
child of the root
        }
    }
}

// Function to read data from the file and build the BST
Node* buildBST() {

    //open the file and check for errors
    ifstream inputFile("bd-dec22-births-deaths-by-region.txt");
    if (!inputFile) {
        cerr << "Error opening file." << endl;
        return nullptr;
    }
}

```

```

    string line;
    bool FirstLineSkipped = false; // Flag to track whether the first line has
    been skipped
    Node* root = nullptr; // root that points to the struct Node is given
    null value

    while (getline(inputFile, line)) {
        if (!FirstLineSkipped) {
            FirstLineSkipped = true;
            continue; // skip first line
        }

        stringstream ss(line); //read and write in string line
        string token; // data from file to be read
        string tokens[4];

        int index = 0; //number of commas to be read

        while (getline(ss, token, ',')) { //read row of file and return
data(token) until it reaches a comma
            tokens[index++] = token; //store data from between the commas of each
row in tokens[]
            if (index >= 4) {
                break; // Avoid accessing out of bounds
            }
        }

        if (index != 4) {
            cerr << "Error: Invalid data format in line: " << line << endl;
            continue; //incorrect amount of commas
        }

        try {
            //store read data in struct rowData
            Row rowData;
            rowData.Period = stoi(tokens[0]);
            rowData.Birth_Death = (tokens[1] == "Births"); // Set true if
"Births", false otherwise
            rowData.Region = tokens[2];
            rowData.Count = stoi(tokens[3]);

            insertNode(root, rowData); //insert the read data into a new node
        } catch (const exception& e) {

```

```

        cerr << "Error: Invalid integer conversion in line: " << line <<
endl;
        continue; //catch errors when converting to integers
    }
}

inputFile.close(); //close file
return root; //return root that points to struct Node
}

// Function to traverse and print the BST in inorder with region headers
void inorderTraversal(Node* root, string& currentRegion) {
    if (root != nullptr) {
        inorderTraversal(root->left, currentRegion); // Traverse left subtree

        // If the region changes, print it as a header
        if (root->data.Region != currentRegion) {
            currentRegion = root->data.Region;
            cout << "\n\nREGION: " << currentRegion << "\n" << endl;
        }

        // Print period and count
        cout << "Period: " << root->data.Period << ", Count: " << root-
>data.Count << endl;

        inorderTraversal(root->right, currentRegion); // Traverse right subtree
    }
}

// Function to search for the number of births for a specific time period and
region
int searchBirthCount(Node* root, int period, const string& region) {
    // Compare the period and region with the current node's data
    if (root == nullptr) {
        // If the root is null, the data doesn't exist
        return -1;
    } else if (root->data.Region == region && root->data.Period == period) {
        // If found, return the count
        return root->data.Count;
    } else if (root->data.Region > region || (root->data.Region == region &&
root->data.Period > period)) {
        // If the target period and region are smaller, search in the left
subtree
        return searchBirthCount(root->left, period, region);
    } else {

```

```

        // If the target period and region are greater, search in the right
        subtree
        return searchBirthCount(root->right, period, region);
    }
}

// Function to modify the number of births for a specific time period and region
void modifyBirthCount(Node* root, int period, const string& region, int newCount)
{
    // If the root is null, the data doesn't exist
    if (root == nullptr) {
        cout << "Data not found." << endl;
        return;
    }

    // Compare the period and region with the current node's data
    if (root->data.Region == region && root->data.Period == period) {
        // If found, modify the count
        root->data.Count = newCount;
        cout << "Number of births for period " << period << " in " << region << "
        modified to " << newCount << endl;
    } else if (root->data.Region > region || (root->data.Region == region &&
    root->data.Period > period)) {
        // If the target period and region are smaller, search in the left
        subtree
        modifyBirthCount(root->left, period, region, newCount);
    } else {
        // If the target period and region are greater, search in the right
        subtree
        modifyBirthCount(root->right, period, region, newCount);
    }
}

// Function to merge two subtrees
Node* mergeSubtrees(Node* left, Node* right) {
    // If one of the subtrees is empty, return the other subtree
    if (left == nullptr) return right;
    if (right == nullptr) return left;

    // Find the rightmost node of the left subtree
    Node* temp = left;
    while (temp->right != nullptr) {
        temp = temp->right;
    }
}

```

```

    // Attach the right subtree to the right of the rightmost node of the left
    subtree
    temp->right = right;
    return left;
}

// Function to delete all nodes with the given region from the BST
Node* deleteNode(Node* root, const string& region) {
    // If the tree is empty
    if (root == nullptr) {
        return root;
    }

    // Recursively delete nodes in left and right subtrees
    root->left = deleteNode(root->left, region);
    root->right = deleteNode(root->right, region);

    // If the current node has the region to be deleted, delete it and return its
    child
    if (root->data.Region == region) {
        Node* temp = root;
        root = mergeSubtrees(root->left, root->right);
        delete temp;
    }

    return root;
}

// Function to delete a record based on the region
void deleteRecordByRegion(Node*& root, const string& region) {
    root = deleteNode(root, region);
    cout << "All records with region " << region << " deleted successfully." <<
endl;
}

// Function to display the menu
void displayMenu() {
    cout << "Menu:" << endl;
    cout << "1. Display the BST with inorder traversal." << endl;
    cout << "2. Search for the number of births for a specific time period and
region." << endl;
    cout << "3. Modify the number of births for a specific time period and
region." << endl;
    cout << "4. Delete a record based on the region." << endl;
    cout << "5. Exit the application." << endl;
}

```

```

}

int main() {
    Node* root = buildBST();    //Read file and create BST
    if (root != nullptr) {    //if tree not empty
        cout << "Binary Search Tree built successfully." << endl;

        int choice;
        do {
            displayMenu(); // Display menu options
            cout << "Enter your choice: ";
            cin >> choice;

            //switch case for menu options
            switch (choice) {
                case 1:{
                    cout << "Inorder Traversal:" << endl;
                    string currentRegion = ""; // Initialize with an empty
string
                    inorderTraversal(root, currentRegion);
                    break;}
                case 2:{
                    int searchPeriod, searchCount;
                    string searchRegion;
                    cout << "Enter the period to search: ";
                    cin >> searchPeriod;
                    cin.ignore(); // Ignore newline character
                    cout << "Enter the region to search: ";
                    getline(cin, searchRegion);
                    searchCount = searchBirthCount(root, searchPeriod,
searchRegion);

                    if (searchCount != -1) {
                        cout << "Number of births for period " << searchPeriod <<
" in " << searchRegion << ": " << searchCount << endl;
                    } else {
                        cout << "Data not found." << endl;
                    }
                    break;
                }
                case 3: {
                    int modifyPeriod, newCount;
                    string modifyRegion;
                    cout << "Enter the period to modify: ";
                    cin >> modifyPeriod;
                    cin.ignore(); // Ignore newline character

```

```

        cout << "Enter the region to modify: ";
        getline(cin, modifyRegion);
        cout << "Enter the new count: ";
        cin >> newCount;
        modifyBirthCount(root, modifyPeriod, modifyRegion, newCount);
        break;
    }
    case 4: {
        string deleteRegion;
        cout << "Enter the region to delete: ";
        cin.ignore(); // Ignore newline character
        getline(cin, deleteRegion);
        deleteRecordByRegion(root, deleteRegion);
        break;
    }
    case 5:
        cout << "Exiting the application." << endl;
        break;
    default:
        cout << "Invalid choice. Please enter a number between 1 and
5." << endl;
    }
    } while (choice != 5);
} else {
    cerr << "Failed to build Binary Search Tree." << endl;    //if tree is
empty print error
}
return 0;
}

```

Δομή Row και Node:

Row δηλώνει τα δεδομένα που θα αποθηκεύονται σε κάθε κόμβο του δέντρου.

Node δηλώνει έναν κόμβο του δέντρου με τα δεδομένα τύπου Row και δείκτες σε αριστερό και δεξί υποδέντρο.

Συναρτήσεις για το BST:

createNode: Δημιουργεί ένα νέο κόμβο BST με δεδομένα Row.

insertNode: Εισάγει ένα νέο κόμβο στο BST με βάση το χαρακτηριστικό Region και Period.

Ανάγνωση και δημιουργία BST από αρχείο:

buildBST: Διαβάζει δεδομένα από το αρχείο κειμένου (bd-dec22-births-deaths-by-region.txt) και δημιουργεί το BST με δεδομένα για γεννήσεις (Births).

Αναζήτηση, τροποποίηση και διαγραφή κόμβων:

searchBirthCount: Αναζητά τον αριθμό γεννήσεων για ένα συγκεκριμένο χρονικό διάστημα και περιοχή.

modifyBirthCount: Τροποποιεί τον αριθμό γεννήσεων για ένα συγκεκριμένο χρονικό διάστημα και περιοχή.

deleteRecordByRegion: Διαγράφει όλους τους κόμβους με τη συγκεκριμένη περιοχή από το BST.

Εκτύπωση BST:

inorderTraversal: Εκτελεί ενδοτερατική διάσχιση στο BST και εκτυπώνει τα δεδομένα συμπεριλαμβανομένων επικεφαλίδων περιοχής όταν αλλάζει η περιοχή.

Κύρια συνάρτηση main:

Διαβάζει και δημιουργεί το BST από το αρχείο.

Προσφέρει ένα μενού επιλογών (εμφάνιση δέντρου, αναζήτηση, τροποποίηση, διαγραφή κόμβων, έξοδος).

Παράδειγμα εμφάνισης Μενού επιλογών:

```
Binary Search Tree built successfully.
Menu:
1. Display the BST with inorder traversal.
2. Search for the number of births for a specific time period and region.
3. Modify the number of births for a specific time period and region.
4. Delete a record based on the region.
5. Exit the application.
Enter your choice: █
```

B. Τροποποίηση ΔΔΑ

Για την τροποποίηση του ερωτήματος A, αλλάξαμε στην κατασκευή του δέντρου, την σύγκριση των περιοχών σε σύγκριση αριθμών γεννήσεων. Αλλάξαμε επίσης και τις επιλογές του μενού.

```
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include <vector>
using namespace std;

struct Row {
    int Period;
    bool Birth_Death;    // 0=Death , 1=Birth
    string Region;
    int Count;
};

// Structure of a BST node
struct Node {
    Row data;
    Node* left; //pointer to left subtree
    Node* right; //pointer to right subtree
};

// Function to create a new node that returns itself
Node* createNode(Row data) {
    Node* newNode = new Node();    //make newNode that points to struct Node
    newNode->data = data;           // copy elements of struct row and transfer
    them to struct newNode
    newNode->left = newNode->right = nullptr;    //make left and right subtree
    pointer null
    return newNode;                //return new node
}

// Function to insert a new node into the BST
void insertNode(Node*& root, Row data) {

    // Only insert if the data is for births
    if (!data.Birth_Death) {
```

```

        return;
    }

    if (root == nullptr) {    //if there is no root create one
        root = createNode(data);
        return;
    }

    // Compare the new count with the current node's count to decide where to
insert
    if (data.Count < root->data.Count) {    // if the count I want to insert
is < than the root's
        insertNode(root->left, data);    // then insert it as a left child of
the root

    } else if (data.Count > root->data.Count) {    // if the count I want to insert
is > than the root's
        insertNode(root->right, data);    // then insert it as a right
child of the root

    } else {
        // If region is the same, decide based on period
        if (data.Period < root->data.Period) { // if the period of the region I
want to insert is < than the root's
            insertNode(root->left, data);    // then insert it as a left child
of the root

        } else {    // if the period of the region I
want to insert is > than the root's
            insertNode(root->right, data);    // then insert it as a right
child of the root
        }
    }
}

// Function to read data from the file and build the BST
Node* buildBST() {

    //open the file and check for errors
    ifstream inputFile("bd-dec22-births-deaths-by-region.txt");
    if (!inputFile) {
        cerr << "Error opening file." << endl;
        return nullptr;
    }
}

```

```

    string line;
    bool FirstLineSkipped = false;    // Flag to track whether the first line
has been skipped
    Node* root = nullptr;            // root that points to the struct Node is
given null value

    while (getline(inputFile, line)) {
        if (!FirstLineSkipped) {
            FirstLineSkipped = true;
            continue;    // skip first line
        }

        stringstream ss(line);    //read and write in string line
        string token;    // data from file to be read
        string tokens[4];

        int index = 0;    //number of commas to be read

        while (getline(ss, token, ',')) { //read row of file and return
data(token) until it reaches a comma
            tokens[index++] = token; //store data from between the commas of each
row in tokens[]
            if (index >= 4) {
                break; // Avoid accessing out of bounds
            }
        }

        if (index != 4) {
            cerr << "Error: Invalid data format in line: " << line << endl;
            continue; //incorrect amount of commas
        }

        try {
            //store read data in struct rowData
            Row rowData;
            rowData.Period = stoi(tokens[0]);
            rowData.Birth_Death = (tokens[1] == "Births"); // Set true if
"Births", false otherwise
            rowData.Region = tokens[2];
            rowData.Count = stoi(tokens[3]);

            insertNode(root, rowData); //insert the read data into a new node
        } catch (const exception& e) {

```

```

        cerr << "Error: Invalid integer conversion in line: " << line <<
endl;
        continue; //catch errors when converting to integers
    }
}

inputFile.close(); //close file
return root; //return root that points to struct Node
}

void displayMenu() {
    cout << "Menu:" << endl;
    cout << "1. Search for Region/Regions with the minimum Count of Births." <<
endl;
    cout << "2. Search for Region/Regions with the maximum Count of Births." <<
endl;
    cout << "3. Exit." << endl;
}

// Function to find the minimum count in the tree
Row findMin(Node* root) {
    if (root == nullptr) {
        throw runtime_error("Tree is empty"); // If the root is null, print
error: tree is empty
    }

    Node* current = root; // set the root to be the current root
    while (current->left != nullptr) { // while the left sub tree of the
current root points somewhere
        current = current->left; // set the root of the left subtree as
current root
    }
    return current->data; //return the data of the node with the
min count
}

// Function to find the maximum count in the tree
Row findMax(Node* root) {
    if (root == nullptr) {
        throw runtime_error("Tree is empty"); // If the root is null, throw
error
    }

    Node* current = root; // set the root to be the current root

```

```

        while (current->right != nullptr) {    // while the right sub tree of the
current root points somewhere
            current = current->right;          // set the root of the right subtree
as current root
        }
        return current->data;                  //return the data of the node with the
max count
    }

int main() {
    Node* root = buildBST();    //Read file and create BST
    if (root != nullptr) {      //if tree not empty
        cout << "Binary Search Tree built successfully." << endl;

        int choice;
        do {
            displayMenu();        // Display menu options
            cout << "Enter your choice: ";
            cin >> choice;

            //switch case for menu options
            switch (choice) {
                case 1: {
                    try {
                        Row minRow = findMin(root); //give minRow the return
value of findMin

                        //print the data of the node with the min count
                        cout << "Period with minimum Count of Births: " <<
minRow.Period << endl;

                        cout << "Region: " << minRow.Region << endl;
                        cout << "Count: " << minRow.Count << endl;

                    } catch (const runtime_error& e) { // if there is runtime
error print that error
                        cerr << e.what() << endl;
                    }
                    break;
                }
                case 2: {
                    try {
                        Row maxRow = findMax(root); //give maxRow the return
value of findMax

```

```

        //print the data of the node with the max count
        cout << "Period with maximum Count of Births: " <<
maxRow.Period << endl;

        cout << "Region: " << maxRow.Region << endl;
        cout << "Count: " << maxRow.Count << endl;

        } catch (const runtime_error& e) { // if there is runtime
error print that error
            cerr << e.what() << endl;
        }
        break;
    }
    case 3:
        cout << "Exiting the application." << endl;
        break;
    default:
        cout << "Invalid choice. Please enter a number between 1 and
3." << endl;
    }
    } while (choice != 3);
} else {
    cerr << "Failed to build Binary Search Tree." << endl;    //if tree is
empty print error
}
return 0;
}

```

Δομή Row και Node:

Row αποθηκεύει τα δεδομένα για κάθε εγγραφή, όπως περίοδος, τύπος (γέννηση ή θάνατος), περιοχή και αριθμός.

Node αποτελεί τον κόμβο του BST που περιέχει τα δεδομένα τύπου Row και δείκτες στα αριστερά και δεξιά παιδιά του.

Συναρτήσεις για το BST:

createNode: Δημιουργεί ένα νέο κόμβο BST με δεδομένα Row.

insertNode: Εισάγει ένα νέο κόμβο στο BST, με βάση τον αριθμό γεννήσεων (Count). Αν υπάρχει ισοτιμία στον αριθμό γεννήσεων, τότε αποφασίζει με βάση την περίοδο.

Ανάγνωση και δημιουργία BST από αρχείο:

buildBST: Διαβάζει δεδομένα το αρχείο κειμένου και δημιουργεί το BST με δεδομένα για γεννήσεις (και ενδεχομένως θάνατους).

Αναζήτηση του κόμβου με τον ελάχιστο αριθμό γεννήσεων:

findMin: Βρίσκει και επιστρέφει τον κόμβο με τον ελάχιστο αριθμό γεννήσεων στο BST.

Αναζήτηση του κόμβου με τον μέγιστο αριθμό γεννήσεων:

findMax: Βρίσκει και επιστρέφει τον κόμβο με τον μέγιστο αριθμό γεννήσεων στο BST.

Κύρια συνάρτηση main:

Αναγνωρίζει την είσοδο από τον χρήστη για να εκτελέσει επιλογές αναζήτησης του κόμβου με τον ελάχιστο ή μέγιστο αριθμό γεννήσεων, ή για έξοδο από την εφαρμογή.

Παράδειγμα εμφάνισης Μενού επιλογών:

```
Binary Search Tree built successfully.  
Menu:  
1. Search for Region/Regions with the minimum Count of Births.  
2. Search for Region/Regions with the maximum Count of Births.  
3. Exit.  
Enter your choice: █
```

Γ. Hashing

Για την υλοποίηση του ερωτήματος Γ, αντί για χρήση δυαδικού δέντρου αναζήτησης, τροποποιήσαμε τον κώδικα του ερωτήματος Α ώστε να χρησιμοποιεί hashing με αλυσίδες. Κάναμε επίσης αλλαγές στο μενού επιλογών ώστε να υλοποιεί τα ζητούμενα της εκφώνησης.

```
#include <iostream>  
#include <fstream>  
#include <string>  
#include <sstream>  
#include <vector>  
#include <list>  
#include <limits>
```



```

using namespace std;

// Define a structure to hold data for each row in the file
struct Row {
    int Period;           // Period of data
    bool Birth_Death;     // Flag indicating if the data is for births (true) or
    deaths (false)
    string Region;        // Region for the data
    int Count;            // Number of births or deaths
};

// Hash table class with chaining
class HashTable {
private:
    vector<list<Row>> table; // Vector of lists for the hash table
    int size;               // Size of the hash table

    // Hash function to calculate the index in the table based on the region
    int hashFunction(const string& region) {
        int sum = 0;
        for (char ch : region) {
            sum += static_cast<int>(ch);
        }
        return sum % size;
    }

public:
    // Constructor to initialize hash table with given size
    HashTable(int s) : size(s), table(s) {}

    // Function to insert a row into the hash table
    void insert(Row data) {
        if (!data.Birth_Death) return; // Only insert if the data is for births
        int index = hashFunction(data.Region);
        table[index].push_back(data);
    }

    // Function to search for a row in the hash table
    Row* search(int period, const string& region) {
        int index = hashFunction(region);
        for (auto& row : table[index]) {
            if (row.Period == period && row.Region == region) {
                return &row;
            }
        }
    }
}

```

```

        return nullptr;
    }

    // Function to update a row in the hash table
    bool update(int period, const string& region, int newCount) {
        int index = hashFunction(region);
        for (auto& row : table[index]) {
            if (row.Period == period && row.Region == region) {
                row.Count = newCount;
                return true;
            }
        }
        return false;
    }

    // Function to delete rows by region in the hash table
    void deleteByRegion(const string& region) {
        int index = hashFunction(region);
        table[index].remove_if([&region](const Row& row) {
            return row.Region == region;
        });
    }

    // Function to display all entries in the hash table
    void display() {
        for (int i = 0; i < size; ++i) {
            if (!table[i].empty()) {
                cout << "Bucket " << i << ":\n";
                for (const auto& row : table[i]) {
                    cout << "Region: " << row.Region << ", Period: " <<
row.Period << ", Count: " << row.Count << endl;
                }
                cout << endl;
            }
        }
    }
};

// Function to read data from the file and build the hash table
HashTable buildHashTable(int size) {
    ifstream inputFile("bd-dec22-births-deaths-by-region.txt");
    if (!inputFile) {
        cerr << "Error opening file." << endl;
        return HashTable(size);
    }
}

```

```

string line;
bool firstLineSkipped = false;
HashTable hashTable(size);

while (getline(inputFile, line)) {
    if (!firstLineSkipped) {
        firstLineSkipped = true;
        continue; // Skip the first line
    }

    stringstream ss(line);
    string token;
    string tokens[4];
    int index = 0;

    while (getline(ss, token, ',')) {
        tokens[index++] = token;
        if (index >= 4) break; // Avoid accessing out of bounds
    }

    if (index != 4) {
        cerr << "Error: Invalid data format in line: " << line << endl;
        continue;
    }

    try {
        Row rowData;
        rowData.Period = stoi(tokens[0]);
        rowData.Birth_Death = (tokens[1] == "Births");
        rowData.Region = tokens[2];
        rowData.Count = stoi(tokens[3]);

        hashTable.insert(rowData);
    } catch (const exception& e) {
        cerr << "Error: Invalid integer conversion in line: " << line <<
endl;
        continue;
    }
}

inputFile.close();
return hashTable;
}

```

```

// Function to display the menu
void displayMenu() {
    cout << "Menu:" << endl;
    cout << "1. Display all entries in the hash table." << endl;
    cout << "2. Search for the number of births for a specific time period and
region." << endl;
    cout << "3. Modify the number of births for a specific time period and
region." << endl;
    cout << "4. Delete a record based on the region." << endl;
    cout << "5. Exit the application." << endl;
}

int main() {
    const int tableSize = 11; // Size of the hash table
    HashTable hashTable = buildHashTable(tableSize); // Build the hash table

    cout << "Hash Table built successfully." << endl;

    int choice;
    do {
        displayMenu(); // Display the menu
        cout << "Enter your choice: ";
        cin >> choice; // Get user's choice

        if (cin.fail()) {
            cin.clear(); // Clear the error flag
            cin.ignore(numeric_limits<streamsize>::max(), '\n'); // Ignore invalid
input
            cout << "Invalid choice. Please enter a number between 1 and 5." << endl;
            continue; // Loop again to get a valid choice
        }

        switch (choice) {
            case 1: { // Display all entries in the hash table
                cout << "Displaying all entries in the hash table:" << endl;
                hashTable.display();
                break;
            }
            case 2: { // Search for the number of births for a specific time period
and region
                int searchPeriod;
                string searchRegion;
                cout << "Enter the period to search: ";
                cin >> searchPeriod;

```

```

        cin.ignore(numeric_limits<streamsize>::max(), '\n'); // Ignore
newline character
        cout << "Enter the region to search: ";
        getline(cin, searchRegion);

        Row* result = hashTable.search(searchPeriod, searchRegion);
        if (result) {
            cout << "Number of births for period " << searchPeriod << " in "
<< searchRegion << ": " << result->Count << endl;
        } else {
            cout << "Data not found." << endl;
        }
        break;
    }
    case 3: { // Modify the number of births for a specific time period and
region
        int modifyPeriod, newCount;
        string modifyRegion;
        cout << "Enter the period to modify: ";
        cin >> modifyPeriod;
        cin.ignore(numeric_limits<streamsize>::max(), '\n'); // Ignore
newline character
        cout << "Enter the region to modify: ";
        getline(cin, modifyRegion);
        cout << "Enter the new count: ";
        cin >> newCount;

        if (hashTable.update(modifyPeriod, modifyRegion, newCount)) {
            cout << "Number of births for period " << modifyPeriod << " in "
<< modifyRegion << " modified to " << newCount << endl;
        } else {
            cout << "Data not found." << endl;
        }
        break;
    }
    case 4: { // Delete a record based on the region
        string deleteRegion;
        cout << "Enter the region to delete: ";
        cin.ignore(numeric_limits<streamsize>::max(), '\n'); // Ignore
newline character
        getline(cin, deleteRegion);

        hashTable.deleteByRegion(deleteRegion);
        cout << "All records with region " << deleteRegion << " deleted
successfully." << endl;
    }
}

```

```

        break;
    }
    case 5: // Exit the application
        cout << "Exiting the application." << endl;
        break;
    default:
        cout << "Invalid choice. Please enter a number between 1 and 5." <<
endl;
    }
} while (choice != 5); // Continue the loop until user chooses to exit

return 0;
}

```

Αυτός ο κώδικας αναπτύχθηκε για να διαχειριστεί δεδομένα για γεννήσεις (ή θανάτους) ανά περιοχή και χρονική περίοδο χρησιμοποιώντας έναν πίνακα κατακερματισμού (hash table) με αλυσίδες για την αντιμετώπιση συγκρούσεων.

Δομή Row:

Αποθηκεύει τα δεδομένα για κάθε εγγραφή, όπως περίοδος, τύπος (γέννηση ή θάνατος), περιοχή και αριθμός.

Κλάση HashTable:

Χρησιμοποιείται για την αποθήκευση και διαχείριση των δεδομένων μέσω ενός πίνακα κατακερματισμού με αλυσίδες (vector<list<Row>> table).

Συνάρτηση κατακερματισμού (hashFunction): Υπολογίζει το δείκτη (index) στον πίνακα βάσει της περιοχής (χρησιμοποιώντας το άθροισμα ASCII των χαρακτήρων).

Συναρτήσεις (insert, search, update, deleteByRegion): Επιτρέπουν την εισαγωγή, αναζήτηση, ενημέρωση και διαγραφή εγγραφών από τον πίνακα κατακερματισμού βάσει της περιοχής και της περιόδου.

Συνάρτηση display: Εμφανίζει όλες τις εγγραφές στον πίνακα κατακερματισμού.

Συνάρτηση buildHashTable:

Διαβάζει τα δεδομένα από ένα αρχείο κειμένου και δημιουργεί τον πίνακα κατακερματισμού με τα δεδομένα για γεννήσεις .

Κύρια συνάρτηση main:

Δημιουργεί ένα αντικείμενο HashTable καλώντας την buildHashTable.

Εμφανίζει ένα μενού επιλογών για τον χρήστη (displayMenu).

Ο χρήστης μπορεί να επιλέξει από τις ακόλουθες επιλογές:

Εμφάνιση όλων των εγγραφών στον πίνακα κατακερματισμού.

Αναζήτηση αριθμού γεννήσεων για συγκεκριμένη περίοδο και περιοχή.

Τροποποίηση αριθμού γεννήσεων για συγκεκριμένη περίοδο και περιοχή.

Διαγραφή εγγραφών βάσει της περιοχής.

Έξοδος από την εφαρμογή.

Παράδειγμα εμφάνισης Μενού επιλογών:

```
Hash Table built successfully.  
Menu:  
1. Display all entries in the hash table.  
2. Search for the number of births for a specific time period and region.  
3. Modify the number of births for a specific time period and region.  
4. Delete a record based on the region.  
5. Exit the application.  
Enter your choice: █
```

Ενοποίηση A,B,Γ

Για την ενοποίηση των προηγούμενων ερωτημάτων φτιάξαμε ένα καινούργιο πρόγραμμα που θα ρωτάει με μενού τον χρήστη με ποιον τρόπο θέλει να φορτώσει τα δεδομένα. Για την επιλογή φόρτωσης σε δυαδικό δέντρο αναζήτησης, ρωτάει το πρόγραμμα αν θέλουμε διάταξη των κόμβων με βάση την περιοχή ή τον αριθμό γεννήσεων.

```
#include <iostream>  
#include <cstdlib> // For system() function  
  
using namespace std;  
  
// Function prototypes  
void displayMenu();
```

```

void loadIntoBST();
void loadIntoHashing();

// Function to display the main menu
void displayMenu() {
    cout << "Select the data structure type for loading the file:" << endl;
    cout << "1. Binary Search Tree (BST)" << endl;
    cout << "2. Hashing with chaining" << endl;
    cout << "3. Exit Program" << endl; // Option to exit the program
    cout << "Choice: ";
}

// Function to load data into Binary Search Tree (BST)
void loadIntoBST() {
    int choice;
    cout << "Select the type of Binary Search Tree (BST):" << endl;
    cout << "1. Load based on REGION" << endl;
    cout << "2. Load based on BIRTH COUNT" << endl;
    cout << "3. Back to main menu" << endl; // Option to go back to main menu
    cout << "4. Exit Program" << endl; // Option to exit the program
    cout << "Choice: ";
    cin >> choice;

    string filename;
    switch (choice) {
        case 1:
            filename = "makeBST-A.exe"; // Assuming the executable file is named
makeBST-A.exe
            break;
        case 2:
            filename = "makeBST-B.exe"; // Assuming the executable file is named
makeBST-B.exe
            break;
        case 3:
            return; // Return to main menu
        case 4:
            exit(0); // Exit the program
        default:
            cout << "Invalid choice. Please try again." << endl;
            return;
    }

    // Execute the selected file
    cout << "Executing " << filename << "..." << endl;
    system(filename.c_str());
}

```



```

}

// Function to load data into Hashing with chaining
void loadIntoHashing() {
    string filename = "Hashing.exe"; // Assuming the executable file is named
    Hashing.exe

    // Execute the selected file
    cout << "Executing " << filename << "..." << endl;
    system(filename.c_str());
}

int main() {
    int choice;
    do {
        displayMenu();
        cin >> choice;

        switch (choice) {
            case 1:
                loadIntoBST();
                break;
            case 2:
                loadIntoHashing();
                break;
            case 3:
                exit(0); // Exit the program
            default:
                cout << "Invalid choice. Please try again." << endl;
                break;
        }
    } while (choice != 3);

    return 0;
}

```

Αυτός ο κώδικας υλοποιεί ένα απλό μενού για την επιλογή δομής δεδομένων για φόρτωση δεδομένων από αρχείο.

Συνάρτηση displayMenu:

Εμφανίζει τις επιλογές για τον τύπο της δομής δεδομένων που θέλει ο χρήστης να φορτώσει τα δεδομένα από αρχείο.

Ο χρήστης μπορεί να επιλέξει ανάμεσα σε BST, Hashing με αλυσίδες ή να αποχωρήσει από το πρόγραμμα.

Συναρτήσεις loadIntoBST και loadIntoHashing:

loadIntoBST: Παρουσιάζει επιλογές για την εκτέλεση δύο διαφορετικών εκτελέσιμων αρχείων (makeBST-A.exe και makeBST-B.exe) που φορτώνουν δεδομένα σε ένα Binary Search Tree (BST) ανάλογα με την επιλογή του χρήστη.

loadIntoHashing: Εκτελεί ένα εκτελέσιμο αρχείο Hashing.exe που φορτώνει δεδομένα σε μια δομή Hashing με αλυσίδες.

Συνάρτηση main:

Ένας βρόχος do-while χρησιμοποιείται για την εμφάνιση και την επεξεργασία του μενού.

Επιλογές:

Ο χρήστης μπορεί να επιλέξει να φορτώσει σε BST (επιλογή 1) ή σε Hashing (επιλογή 2).

Υπάρχει επίσης η επιλογή για έξοδο από το πρόγραμμα (επιλογή 3).

Εάν η επιλογή δεν είναι έγκυρη, εκτυπώνεται αντίστοιχο μήνυμα λάθους.

Παράδειγμα εμφάνισης Μενού επιλογών:

```
Select the data structure type for loading the file:
1. Binary Search Tree (BST)
2. Hashing with chaining
3. Exit Program
Choice: 1
Select the type of Binary Search Tree (BST):
1. Load based on REGION
2. Load based on BIRTH COUNT
3. Back to main menu
4. Exit Program
Choice: █
```

```
Select the data structure type for loading the file:
1. Binary Search Tree (BST)
2. Hashing with chaining
3. Exit Program
Choice: 2
Executing Hashing.exe...
Hash Table built successfully.
Menu:
1. Display all entries in the hash table.
2. Search for the number of births for a specific time period and region.
3. Modify the number of births for a specific time period and region.
4. Delete a record based on the region.
5. Exit the application.
Enter your choice: █
```

ΤΕΛΟΣ ΑΝΑΦΟΡΑΣ