University of Žilina

Faculty of Management Science and Informatics

UNIVERSITY OF ŽILINA
Faculty of Management Science
and Informatics

Semestral Project Algorithms and Data Structures

Done by:

[Rodrigo Lourenço]

Orientation by:

Mr. Michal Mrena
Mr. Michal Varga

May 25, 2025
2024/2025

Appreciations

The completion of this work/project, as well as the vast majority of my academic life, would not be possible without the help of some people, so i cannot let them go unnoticed at this time.

First of all, i would like to thank Mr.Michal Mrena and Mr.Michal Varga, for all the knowledge transmitted in this Algorithms and Data Structures Course throughout the semester and also for all the help provided throughout the semester.

I would also like to thank my family, who despite being far away, i know that they are with me and are always willing to support me throughout this experience that ends up being Erasmus.

These are the main keys and elements for this work to be completed within the deadline proposed by the professor and i hope that it will be completed successfully.

Contents

List of Figures

List of Code Snippets

Chapter 1

Introduction

1.1 Framing

This documentation was made in the context of the Algorithms and Data Structures course included in the Erasmus + Courses Program of the Faculty of Management Science and Informatics (FRI) of the University of Žilina (UNIZA), taught by Professor Mr.Michal Mrena in the pratical component and by Professor Mr.Michal Varga on the theoretical component.

This same documentation aims to describe the work carried out and presented in advance on May 25, 2025.

The work proposal and the objectives in order to carry out the final project were made available through the *Microsoft Teams* platform of the University of Žilina (UNIZA).

The main objective of this project was to implement a robust and efficient solution for handling population data spanning five years (2020–2024), organized across multiple hierarchical levels of territorial units in Austria, from municipalities up to the national level.

1.2 Motivation

This work is exciting and important as it allows us to consolidate some the material covered in both practical and theoretical classes. Its completion makes it possible to explore and apply the knowledge acquired throughout this semester in the Algorithms and Data Structures course at different levels. The application of these concepts and knowledge acquired will be part of the management and execution of future work.

1.3 Objectives

In order to carry out this work more effectively, the tasks were splited by 4 parts:

- Level 1 - Load and manage flat population data for municipalities, implementing generic filtering capabilities based on predicates (substring matching and population thresholds).
- Level 2 - Build a hierarchical structure of territorial units (country, geographical divisions, states, regions, municipalities), accurately accumulating and summarizing population data across all levels.
- Level 3 - Extend functionality by adding efficient data retrieval through tables optimized for searching territorial units by their name and type, significantly enhancing lookup speed.
- Level 4 - Implement universal sorting capabilities and advanced filtering operations within selected parts of the hierarchical structure, providing sorted outputs based on alphabetical order (including handling diacritics) and population size for specified criteria. The project's implementation strictly adheres to guidelines regarding manual memory management, performance efficiency, and careful handling of textual data, especially concerning diacritics in UTF-8 encoding.
- Final Documentation - Explanation of all the work done according to the requisites that are present in the document SP.Rules. The documentation was made using the language $\LaTeX$and using the website `overleaf.com`

1.4 Document Organization

In order to better understand and comprehend the work, this documentation is organized into 9 different chapters:

1. **Chapter 1: Introduction**
2. **Chapter 2: Level 1: Flat Data Loading and Filtering**
3. **Chapter 3: Level 2: Hierarchy of Territorial Units**
4. **Chapter 4: Level 3: Efficient Search Tables**
5. **Chapter 5: Sorting and Additional Filtering**

6. **Chapter 6: Application Design and UML Diagrams**
7. **Chapter 7: Programmer's Guide**
8. **Chapter 8: User Guide**
9. **Chapter 9: Conclusions**

This structure aims to guide the reader in a clear and logical way, providing an in-depth understanding of the work and ultimately facilitating navigation through the content presented.

Chapter
2

Level 1: Flat Data Loading and Filtering

2.1 Description of the Data Loaded

Level 1 of the project involves loading and managing population data for Austrian municipalities. The data is sourced from five separate CSV files (`2020.csv`, `2021.csv`, `2022.csv`, `2023.csv`, `2024.csv`).

Each file contains:

- Municipality Name
- Municipality Code
- Male Population
- Female Population

The data for each municipality is merged and organized into a unified data structure that captures a five-year population history.

2.2 Data Structures Used

Two primary data structures are used at this level, both developed and implemented in labs:

- **Vector (dynamic array)**:

  Chosen for storing municipality records due to its simplicity and efficient direct access and iteration capabilities.

- **Map (associative map)**:

  Utilized for efficient lookup operations, specifically mapping municipality codes to their corresponding index positions within the `Vector`.

2.3 Implementation Details

2.3.1 FlatMunicipality Struct

Data from CSV files is encapsulated into the following structure:

```
struct FlatMunicipality {
    std::string name;
    std::string code;
    int male[5];
    int female[5];
}
```

Code Snippets 2.1: FlatMunicipality Struct

Each municipality stores its population counts separately for each year.

2.3.2 Data Loading Algorithm

The algorithm performs the following steps:

1. Opens each CSV file corresponding to years 2020–2024.
2. Reads data line-by-line, parsing values for name, code, male, and female populations.
3. Cleans municipality codes by removing non-alphanumeric characters using `cleancode`.
4. Uses a `Map` to efficiently track previously seen municipality codes and their respective indices in the `Vector`.
5. If a municipality record already exists, it updates the existing entry. Otherwise, it creates a new entry.

2.3.3 Generic Filtering Algorithm

The filtering functionality is encapsulated as a generic template algorithm that accepts predicates to select municipalities according to specified conditions:

```
Vector<FlatMunicipality> filter(const Vector<FlatMunicipality>& data,
    Pred predicate) {
    Vector<FlatMunicipality> result;
    for (size_t i = 0; i < data.size(); ++i) {
        if (predicate(data[i])) {
            result.push_back(data[i]);
        }
    }
    return result;
}
```

Code Snippets 2.2: Generic Filtering Algorithm

2.3.4 Predicates Used

Three specific predicates were implemented as lambda functions:

- `containsStr`:Filters municipalities whose names contain a specified substring (case-insensitive).
- `hasMaxResidents`:Filters municipalities whose total population in a specific year does not exceed a given threshold.
- `hasMinResidents`: Filters municipalities whose total population in a specific year meets or exceeds a given threshold.

Lambda functions ensure flexibility, enabling predicate modification without changing the algorithm itself.

Chapter

3

Level 2: Hierarchy of Territorial Units

3.1 Overview of Hierarchy Structure

In Level 2, the project enhances the flat data from Level 1 by structuring it into a hierarchical representation of Austrian territorial units. The hierarchy includes several clearly defined levels:

- **Root**: An imaginary node representing Austria.
- **Geographical Divisions**: Nodes representing the Eastern, Western, and Southern regions of Austria.
- **Federal States**: Nodes representing each Austrian federal state.
- **Regions**: Nodes within each federal state.
- **Municipalities**: Leaf nodes representing individual municipalities.

3.2 Data Structures and Definitions

3.2.1 TerritorialUnit Struct

Stores essential data for each territorial unit:

```
struct TerritorialUnit {
    std::string name;
    std::string code;
    std::string type; // Country, GeoDiv, State, Region, Municipality
    Map<std::string, std::pair<int, int>> popByYear; // Population data:
        year    (male, female)
};
```

Code Snippets 3.1: TerritorialUnit Struct

3.2.2 HierarchyNode Struct

Represents nodes in the hierarchy:

```
struct HierarchyNode {
    TerritorialUnit unit;
    Vector<HierarchyNode*> children;
    HierarchyNode* parent = nullptr;

    HierarchyNode(const TerritorialUnit& u) : unit(u) {}

    ~HierarchyNode() {
        for (size_t i = 0; i < children.size(); ++i)
            delete children[i];
    }
};
```

Code Snippets 3.2: HierarchyNode Struct

This struct supports the hierarchical structure with pointers linking parents and children.

3.3 Implementation Details

3.3.1 Loading Hierarchical Data

Data is loaded in multiple steps:

1. **`loadRegions` function**: Loads regions, federal states, and geographical divisions from `country.csv`. Nodes are created and linked to their respective parents based on their code structure.

2. **`loadMunicipalities` function**: Loads municipality data from `municipalities.csv`, linking them to their appropriate region nodes using a lookup table.

3. **Population Data Loading**: Population data from Level 1 CSV files (2020–2024) is loaded and aggregated into the hierarchy nodes through the `loadPopData` function.

3.3.2 Population Accumulation Algorithm

Population data is accumulated bottom-up using a recursive post-order traversal (`accumulate` function):

- Each node sums the male and female populations of its children, cumulatively storing these totals to allow easy access and summary at higher levels of the hierarchy.

3.4 Hierarchy Iterator

A manual forward iterator allows users to navigate through the hierarchy interactively. Users can perform the following operations:

- Move to a child node.
- Move back to a parent node.
- List all immediate children of the current node.
- Show a summary of the cumulative population at the current node.

```
[Navigate] Burgenland (State)
1) List Children
2) Move to Child
3) Move to Parent
4) Show Summary
0) Back
```

Code Snippets 3.3: Interactive Navigation Example:

3.5 Justification for Data Structures Chosen

- **Vector**: Ideal for managing children of hierarchy nodes due to its simplicity and efficient sequential access.
- **HashMap**: Used for quick lookup and linking of nodes based on their unique codes, essential for efficient hierarchical linking and data merging.

- **Map**: Provides structured access to population data by year within each territorial unit, ensuring ordered and efficient retrieval of population statistics.

Chapter

4

Level 3: Efficient Search Tables

4.1 Introduction to Level 3

In Level 3, the functionality developed in previous levels is expanded to incorporate efficient searching mechanisms. Specifically, the project introduces specialized hash-based tables enabling fast lookup of territorial units based on their names and types. This significantly improves user experience and operational performance by reducing search time complexity.

4.2 Data Structures and Definitions

The following hash tables (`HashMap`) are introduced, one for each type of territorial unit:

- Country Table (`countryTable`)
- Geographical Division Table (`geoDivTable`)
- Federal State Table (`stateTable`)
- Region Table (`regionTable`)
- Municipality Table (`municipalityTable`)

Each table maps unit names to vectors of pointers to `HierarchyNode` instances:

```
HashMap<std::string, Vector<HierarchyNode*>> countryTable;
HashMap<std::string, Vector<HierarchyNode*>> geoDivTable;
HashMap<std::string, Vector<HierarchyNode*>> stateTable;
HashMap<std::string, Vector<HierarchyNode*>> regionTable;
HashMap<std::string, Vector<HierarchyNode*>> municipalityTable;
```

Code Snippets 4.1: HierarchyNode instances

4.3 Implementation Details

4.3.1 Building Search Tables

Search tables are constructed recursively, traversing the hierarchical structure created in Level 2. The key algorithm is the `buildTables` function:

```
static void buildTables(
    HierarchyNode* node,
    HashMap<std::string, Vector<HierarchyNode*>>& countryT,
    HashMap<std::string, Vector<HierarchyNode*>>& geoDivT,
    HashMap<std::string, Vector<HierarchyNode*>>& stateT,
    HashMap<std::string, Vector<HierarchyNode*>>& regionT,
    HashMap<std::string, Vector<HierarchyNode*>>& muniT)
{
    const std::string& t = node->unit.type;
    if (t == "Country") countryT[node->unit.name].push_back(node);
    else if (t == "GeoDiv") geoDivT[node->unit.name].push_back(node);
    else if (t == "State") stateT[node->unit.name].push_back(node);
    else if (t == "Region") regionT[node->unit.name].push_back(node);
    else if (t == "Municipality") muniT[node->unit.name].push_back(node)
        ;

    for (size_t i = 0; i < node->children.size(); ++i) {
        buildTables(node->children[i],
                    countryT, geoDivT, stateT, regionT, muniT);
    }
}
```

Code Snippets 4.2: buildTables function

Each entry in these tables allows rapid retrieval based on a given type and name.

4.3.2 Searching by Type and Name

A simple interface allows the user to perform lookups based on exact name and specified type. The operation:

1. Prompts the user to input the type (`Country, GeoDiv, State, Region, Municipality`) and the exact name of the territorial unit.
2. Searches the corresponding hash table.
3. Quickly retrieves and presents population summary information from matching nodes.

4.4 Handling Duplicates and Collisions

Due to potential duplicate territorial unit names within different branches of the hierarchy, vectors of pointers are used to store all matching nodes. This ensures that all relevant results are returned upon a query, effectively managing potential collisions by collecting multiple entries per key.

4.5 Justification for Data Structure Choices

- HashMap (hash table):
  - Chosen for its superior average-case complexity in search operations.
  - Ideal for rapid access based on unique keys (territorial unit names).
- Vector:
  - Used in combination with hash tables to conveniently handle duplicate names and store multiple node pointers efficiently.

Chapter

5

Level 4: Sorting and Additional Filtering

5.1 Introduction to Level 4

Level 4 introduces advanced capabilities: the application of generic sorting algorithms and enhanced filtering within the hierarchical structure created in previous levels. Users can now extract subsets of territorial units from any point in the hierarchy, filter these subsets based on specified criteria, and sort the results alphabetically or by population size.

5.2 Key Functionalities Implemented

The primary new functionalities at Level 4 include:

- **Flattening Subtrees**: Extracting all territorial units beneath a selected node into a flat data structure.
- **Advanced Filtering**: Applying existing predicates (`substring matching`, `population thresholds`) to subsets of hierarchical data.
- **Generic Sorting Algorithm**: Sorting extracted data alphabetically (considering diacritics) or numerically by population criteria.

5.3 Data Structures and Definitions

Level 4 operations leverage previously defined structures (`Vector`, `HierarchyNode`, `TerritorialUnit`) and introduce comparators for sorting purposes:

- **Alphabetical comparator** (`compareAlphabetical`): Uses locale-aware comparison to handle strings containing diacritics accurately.
- **Population comparator** (`comparePopulation`): Sorts territorial units numerically based on specified population parameters (year, gender).

5.4 Implementation Details

5.4.1 Flattening Hierarchy

The function `flatten` converts a subtree rooted at a selected node into a flat structure for further processing:

```
static void flatten(HierarchyNode* node, Vector<TerritorialUnit>& out) {
    out.push_back(node->unit);
    for (size_t i = 0; i < node->children.size(); ++i)
        flatten(node->children[i], out);
}
```

Code Snippets 5.1: Flattening Hierarchy

5.4.2 Advanced Filtering

Using the filtering mechanism from Level 1, Level 4 allows predicates like substring search and population-based filters to be applied specifically within selected parts of the hierarchy:

- Name Substring
- Maximum Population
- Minimum Population

5.4.3 Universal Sorting Algorithm

A generic insertion-sort algorithm (chosen for its simplicity and clear demonstration purposes) sorts filtered territorial units:

```
static void sortData(Vector<TerritorialUnit>& data,
    std::function<int(const TerritorialUnit&, const TerritorialUnit&)>
        cmp)
{
    for (size_t i = 1; i < data.size(); ++i) {
        TerritorialUnit key = data[i];
        size_t j = i;
        while (j > 0 && cmp(key, data[j - 1]) < 0) {
            data[j] = data[j - 1];
            --j;
        }
        data[j] = key;
    }
}
```

Code Snippets 5.2: Universal Sorting Algorithm

5.4.4 Comparator Implementations

- Alphabetical Comparator (Locale-aware for diacritics handling):

```
#ifdef _WIN32
    std::locale loc("Slovak_Slovakia.1250");
#else
    std::locale loc("sk_SK.UTF-8");
#endif
cmp = [loc](const TerritorialUnit& a, const TerritorialUnit& b) {
    auto const& coll = std::use_facet<std::collate<char>>(loc);
    int r = coll.compare(
        a.name.data(), a.name.data() + a.name.size(),
        b.name.data(), b.name.data() + b.name.size());
    return (r < 0 ? -1 : (r > 0 ? 1 : 0));
};
```

Code Snippets 5.3: Alphabetical Comparator

- Population Comparator:

```
cmp = [&](const TerritorialUnit& a, const TerritorialUnit& b) {
    auto ia = a.popByYear.find(year), ib = b.popByYear.find(year);
    int am = 0, af = 0, bm = 0, bf = 0;
    if (ia != a.popByYear.end()) { am = ia->second.first; af = ia
        ->second.second; }
    if (ib != b.popByYear.end()) { bm = ib->second.first; bf = ib
        ->second.second; }
    int va = (sex == "male" ? am : (sex == "female" ? af : (am +
        af)));
    int vb = (sex == "male" ? bm : (sex == "female" ? bf : (bm +
        bf)));
    return (va < vb ? -1 : (va > vb ? 1 : 0));
};
```

Code Snippets 5.4: Population Comparator

5.5 Handling Diacritics

The sorting algorithm respects diacritics by using the standard library's locale-aware comparison (`std::locale` with `std::collate`). This ensures accurate and culturally correct alphabetical sorting, essential for Austrian municipality names.

5.6 Justification of Design Choices

- Insertion Sort:
  - Chosen for simplicity and clarity of the demonstration, clearly illustrating sorting principles.
- Locale-based Comparator:

  to accurately handle alphabetical sorting with diacritics in UTF-8 encoded data.

Chapter 6

Application Design and UML Diagram

6.1 Architectual Overview

The application is structured in four progressive levels. Each level builds on top of the previous, maintaining modularity, code reuse, and a clean separation of responsibilities:

- **Level 1**: Handles flat data loading from CSVs and implements generic filtering algorithms using lambda-based predicates.
- **Level 2**: Constructs a multi-level hierarchy of territorial units (country $\rightarrow$ geoDiv $\rightarrow$ state $\rightarrow$ region $\rightarrow$ municipality) using a custom tree structure.
- **Level 3**: Adds efficient name+type lookup capabilities using hash tables that map names to nodes.
- **Level 4**: Introduces subtree flattening, predicate-based filtering, and comparator-driven sorting on selected parts of the hierarchy.

Each level operates on shared structures like `FlatMunicipality, TerritorialUnit, HierarchyNode` and uses common algorithms such as `filter, flatten, and sortData`.

6.2 Component Roles

- `FlatMunicipality` - Stores a municipality's population data for 5 years (flat data structure).
- `TerritorialUnit` - General structure representing any unit (municipality, region, state, etc.).
- `HierarchyNode` - Tree node containing a TerritorialUnit with parent-child references.
- `Vector<T>` - Dynamic array implementation used for children and data collections.
- `Map<K,V>` - Year-based associative storage for population data.
- `HashMap<K,V>` - Name/type hash tables for fast retrieval (used in Level 3).
- `filter()` - Generic template algorithm to filter any container with a user predicate.
- `sortData()` - Custom sort using insertion sort and a comparator
- `flatten()` - Recursively extracts subtree content to a flat container.

6.3 Application Flow

The diagram below illustrates the logical sequence from startup through user interaction:

1. main() initializes the system.
2. CSV loading gathers municipality and population data (2020–2024).
3. The hierarchy tree is built using the country and municipalities files.
4. The interactive menu allows the user to choose between:
   - Applying flat filters
   - Navigating the hierarchy
   - Searching using name/type
   - Extracting and sorting data from subtrees

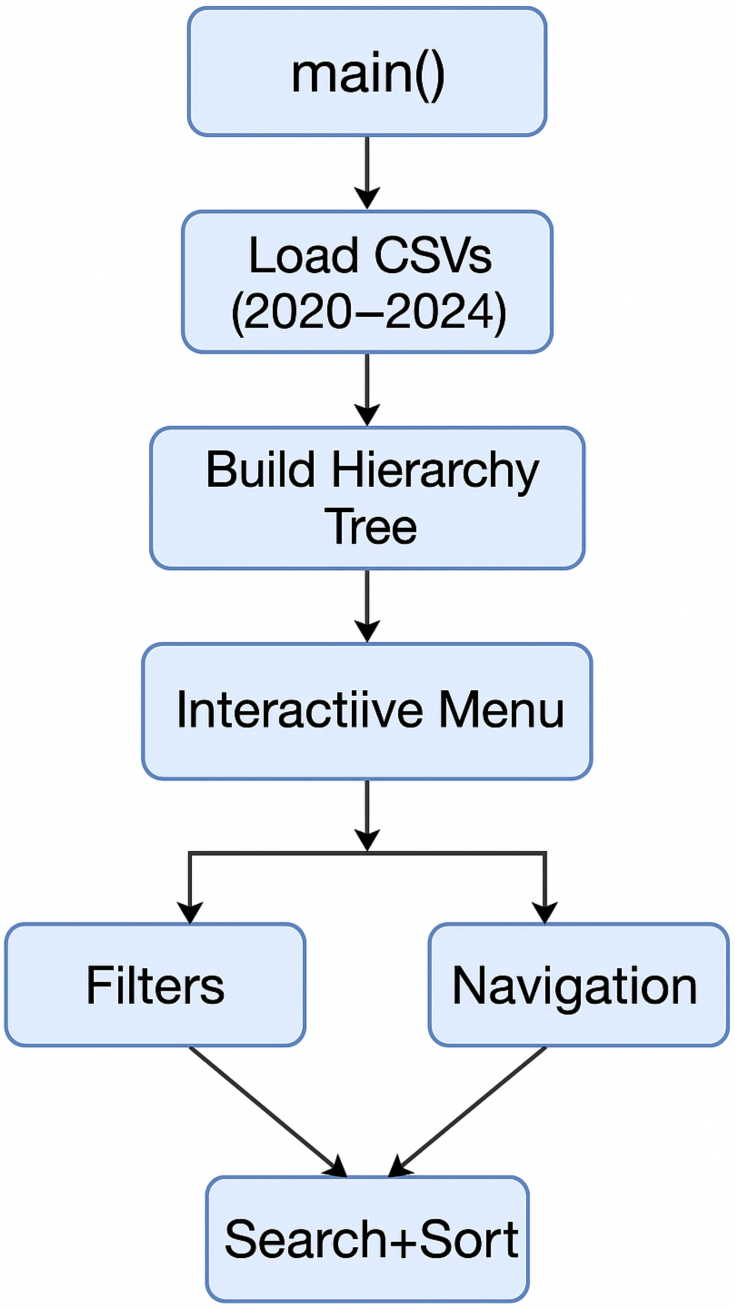

Figure 6.1: Application Design Flowchart

6.4 Internal Class Structure

To model and organize data relationships and behavior, the project uses several interrelated classes. The UML diagram below captures key attributes and their connections:

- `HierarchyNode` composes a `TerritorialUnit` and holds a dynamic array of child nodes (`Vector<HierarchyNode*>`).
- Both `FlatMunicipality` and `TerritorialUnit` are linked to population data.
- Algorithms like `filter`, `flatten`, and `sortData` interact with `Vector`, `Map`, and the hierarchy tree.

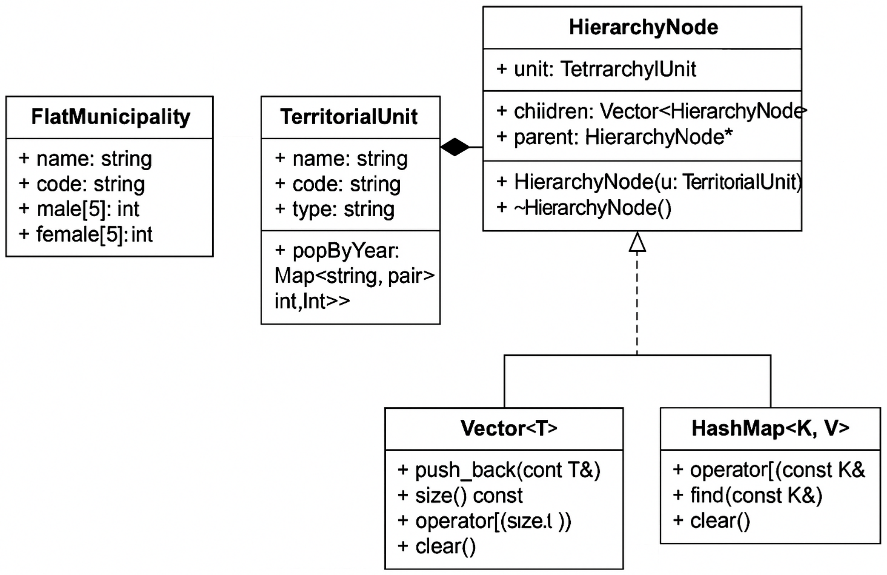

Figure 6.2: UML Diagram of the Internal Class Structure

Chapter 7

Programmer's Guide

This guide provides essential instructions and insights to help programmers understand, run, and extend the software efficiently.

7.1 Compilation and Execution

7.1.1 Prerequisites

- C++ compiler (I used Visual Studio)
- UTF-8 encoding support for proper handling of diacritics
- Data files:
  - `2020.csv, 2021.csv, 2022.csv, 2023.csv, 2024.csv`
  - `country.csv`
  - `municipalities.csv`

7.1.2 Compilation Instructions

Use the following combination (CTRL+Shift+B) in order to build and compile the solution (assuming all required files are in the same directory)

7.1.3 Execution

Run the compiled executable clicking on the green triangle.

7.2 Extending the Filtering Algorithm with New Predicates

The filtering algorithm (`filter`) is generic and utilizes lambda functions to easily add new filtering predicates.
Example of Adding a Predicate:

```cpp
// Example predicate: Filter municipalities starting with a certain
    letter
auto startsWithPredicate = [](const FlatMunicipality& m) {
    return m.name[0] == 'A';
};

auto filteredData = filter(municipalitiesVector, startsWithPredicate);
```

Code Snippets 7.1: Example of Adding a Predicate

New predicates require no modifications to the underlying algorithm.

7.3 Extending the Sorting Algorithm with New Comparators

Sorting is handled via a comparator passed as a lambda function or functional object.
Example of Adding a Comparator:

```cpp
// Comparator: sort by total population descending
auto totalPopulationComparator = [&](const TerritorialUnit& a, const
    TerritorialUnit& b) {
    int popA = a.popByYear["2024"].first + a.popByYear["2024"].second;
    int popB = b.popByYear["2024"].first + b.popByYear["2024"].second;
    return (popA > popB) ? -1 : ((popA < popB) ? 1 : 0);
};

sortData(territorialUnitsVector, totalPopulationComparator);
```

Code Snippets 7.2: Example of Adding a Comparator

This flexible structure simplifies future extensions.

7.4 Memory Management Notes

The application strictly manages memory manually. Key considerations:

- Always pair each `new` with a corresponding `delete`.

- All dynamic allocations in `HierarchyNode` are properly freed using its destructor.
- Use Visual Studio's heap monitor regularly to ensure no memory leaks occur.

Chapter

8

User Guide

This guide instructs general users on operating and navigating the software.

8.1 Launching the Application

Run the executable file, by clicking on the green triangle where the main menu will appear presenting the avaiable functionalities.

8.2 Navigating the Menu

Main Menu Options:

```
/=============== MENU ================/
   [1]  Filter  by Name substring
   [2]  Filter  by Max population
   [3]  Filter  by Min population
   [4]  Hierarchy: Navigate
   [5]  Search  by Type  and Name
   [6]  Filter+Sort  from  Subtree
   [0]  Exit
/=====================================/
```

Code Snippets 8.1: Main Menu Options

Examples of Use:

- Example 1: Filter by Name

   1. Enter 1 from the main menu.
   2. Enter a year (2020–2024).

3. Enter a substring (e.g., "kreutz").
4. View matching municipalities.

- Example 2: Hierarchical Navigation
  1. Select 4 to navigate the hierarchy.
  2. Use provided commands to move through territorial levels (geographical division, state, region, etc.).
  3. Choose to view population summaries or move between nodes.
- Example 3: Search by Type and Name
  1. Select 5 from the menu.
  2. Enter territorial unit type (Country, GeoDiv, State, Region, Municipality).
  3. Enter the exact name of the unit.
  4. View detailed population summaries.
- Example 4: Advanced Filter and Sort
  1. Select 6 from the menu.
  2. Navigate and select a subtree.
  3. Choose filtering criteria (substring, max population, min population).
  4. Choose sorting (alphabetically or by population).
  5. See sorted and filtered results displayed clearly.

8.3 Exiting the Application

- Select option 0 at any menu level to safely exit the application.

Chapter

9 Conclusions

9.1 Principal Conclusions

This documentation presented a structured and comprehensive overview of the Semestral Project for Algorithms and Data Structures, clearly demonstrating the implementation and enhancement of functionalities through four progressively complex levels.
Attention to manual memory management and correct handling of diacritical marks in UTF-8 encoding significantly enhanced reliability and accuracy, ensuring professional-quality results.