Design Document: Data Processing Application

1. **Introduction**

This design document outlines the architecture and functionality of the Data Processing Application. The application aims to perform various operations for working with files, including CSV file reading, data parsing, file creation, file accessing, sequential processing, length-indicated file conversion, indexing, and searching. The project is divided into two parts.

The motivation for this project is to create an efficient system for accessing records in a file structure by using header records. The generation of the header record is done in linear time, as every record in the csv files must be read to generate it. However, once this step is completed, the records in the file structure will be accessible in constant time using a primary key index.

1. **Part 1: Buffer Classes and File Architecture**

The first part of the project is geared towards modifying existing code from project 1 to perform similar processes on zip code data. Data is read from a csv through a buffer. The data is compiled to generate a header file, then the data and header file are stored in a custom file structure defined by the header record.

**2.1 CSVReader Class**

The CSVReader class encompasses the following key aspects:

* **Private Members:**
  + **ZipCSC:** Represents the input CSV file stream used to open and read the CSV file.
  + **Headers:** Stores the column headers from the CSV file.
  + **StateMaximums:** Stores state ID, as well as the maximum locations.
* **Structs:**
  + **Row:** Represents a row of data in the CSV file. This struct stores information for a single row of data in the CSV file, including the ZIP code, name, state, county, latitude, and longitude.
  + **State:** Represents state-related data. This struct stores information related to a state, including the state ID, and the extreme values for latitude and longitude (NorthMost, SouthMost, EastMost, and WestMost rows) within that state.
* **Public Member Functions:**
  + **CSVReader Constructor:** The CSVReader constructor opens a specified CSV file for reading. It takes the file name as a parameter and initializes the ZipCSV object.
  + **isOpen() Function:** The isOpen() function checks if the CSV file is open for reading. It returns true if the file is open and false otherwise.
  + **GetHeaders() Function:** The GetHeaders() function parses and stores the header row of the CSV file. It extracts column headers from the input line and populates the Headers vector with them.
  + **ReadFile() Function:** The ReadFile() function reads and processes the entire CSV file. It starts by reading and storing the header row using the GetHeaders() function, and then it proceeds to read and process each data row of the CSV file.
  + **ParseLine() Function:** The ParseLine() function is responsible for parsing a single data row of the CSV file into a Row object. It updates the Row object with data from the input line.
  + **CheckMaxima() Function:** The CheckMaxima() function checks and updates the StateMaximums map with maximum and minimum values for latitude and longitude based on the input Row.
  + **CompareExtremes() Function:** The CompareExtremes() function compares and updates the maximum and minimum values for latitude and longitude within a state. It determines the extremities for each state based on the input Row.
  + **GetStateMaximums() Function:** The GetStateMaximums() function retrieves a copy of the StateMaximums map, which contains state statistics including maximum and minimum values.
  + **GetStateHeaders() Function:** The GetStateHeaders() function retrieves a copy of the Headers vector, which contains column headers extracted from the CSV file.
  + **close() Function:** The close() function is responsible for closing the CSV file if it is currently open. It ensures that file resources are properly managed.
* **Assumptions**
  + The input CSV file is properly formatted and contains valid data.
  + The CSV file includes a header row that defines column names.
  + Latitude and longitude values are provided in decimal format.
  + The CSV file contains data for multiple states.
  + The CSV file adheres to the format: Zip, Name, State, County, Latitude, Longitude.
  + All rows in the data set contain valid data.
  + The CSV file may be large, so memory usage is a consideration.
  + State statistics, including maximum and minimum values, are calculated and stored for each state in the data.

**2.2 Modifications to CSVReader**

For the purpose of this project additional methods will need to be added to CSVReader.

* **Structs:**
  + **HeaderRecord:** This struct contains data that defines the data file that records will be saved into. Data includes:
    - file name,
    - version,
    - Header record size,
    - count of bytes for each record size integer (if fixed size),
    - size format type {ASCII or binary,
    - primary key index file name, record count,
    - count of fields per record,
    - ordinality (which field serves as the primary key,
    - and for each field: name/ID and type schema (read or write).
  + **Record:** A structure that contains data for a length indicated record. Data includes the records length, zip code fields, and its primary key index.
* **New Members:**
  + **Index:** A map for holding the primary keys of the records
* **New Methods:**
  + **ConvertToLength():** Converts coma delimited records to length indicated records.
  + **WriteToFile():** Writes length indicated records to a data file. (Should be done after the header file is wrote).
  + **ReadFile():** Reads length indicated records from a data file. (Should be done after the header file is read).
  + **GenerateHeaderRecord():** Creates a header record for a data file.
  + **BuildDataFile:** Initializes a data file to read and write length indicated records from/to.

**2.3 Header Record Buffer Class**

The purpose of this class is to read the header record for a data file and parse its fields, as well as to write a header record to a data file.

* **Structs:**
  + **HeaderRecord:** See definition in 2.2.
* **Methods:**
  + **ReadHeaderRecord():** Reads the beginning of a file to pull its header record. Parses the fields of the header record and stores them in a header record struct to be used by other classes/methods.
  + **WriteHeaderRecord():** Takes a header record struct and writes its contents to a data file.
  + **parser():** parses header record data from files.
  + **lengthDecorder():**helps break down the length indicators.

1. **Part 2 Processing command line arguments**

The second part of the project focuses on creating methods to read the header file, and subsequently access data within the data file using information from the header file. The result is a file structure whose records can be accessed from the command line using flags.

* 1. **Command line Reader**
* **Methods:**
  + **Main():** Should be the location of main. Loop will run that prompts the user to give command line inputs and remains running until the user closes the program.
  + ParseCommandLine(): Takes command line input and parses it for flags to determine a course of action.
    - Should parse a flag to search for zip codes.
    - Should parse a flag to display instructions for using the command line.
    - Should parse a flag to indicate which data file to read.
    - Should parse a flag to close the program.
  1. **Primary Key Index**
* **Methods:**
  + **BuildIndex():** If no index exists for a data record,builds an index of primary keys by sequentially accessing records.
  + **ReadIndex():** If an index for a data file exists already, reads its contents to memory.
  + **WriteIndex():** Once the index has been built, write it to a file. If an index already exists, no writing should be done.
  + **SearchIndex():** Searches the index for records with a matching primary key index.
  + **UnpackRecord():** After a record has been found (or not found) using its primary key, existing records are unpacked and displayed with each field labeled. If it is not found, indicate so.