**Project 2 Design Document**

**Group 1**

Myint Aung

Justine Canlas

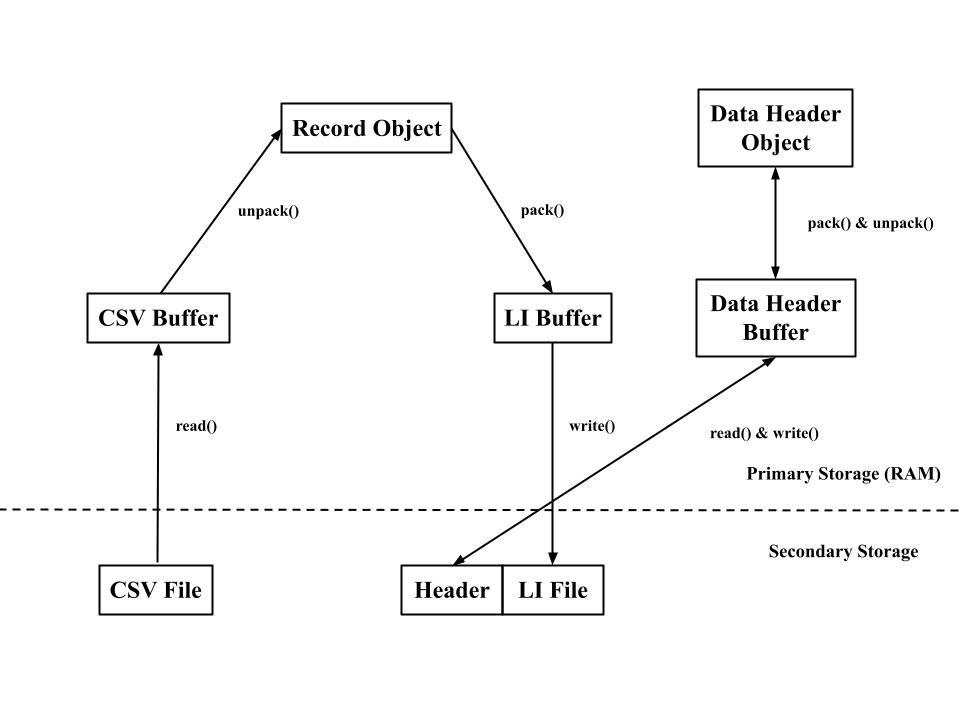
Mariah Coughlin

Jordan Hebler

Anuja Modi

**Requirements**

1. Create a CSV (comma separated) file from this [XLSX file](http://web.stcloudstate.edu/aanda/cs331/w19/us_postal_codes.xlsx).
2. Create a CSV (comma separated) file from this row-*randomized* [XLSX file](http://web.stcloudstate.edu/aanda/cs331/w19/us_postal_codes_ROWS_RANDOMIZED.xlsx).
   * Delete the column of random values before saving as a CSV file
3. Process sequentially a CSV file using a buffer class. {functionality from Group Project 1.0}
   * ensure that Group Project 1.0 works equivalently for the *row-randomized* XLSX file.
4. Convert both CSV files to a file structure format (beginning with a header record) where the fields are still comma separated, but the records are length indicated.
5. Modify your **buffer class** to read and unpack a Zip Code Record from a *length-indicated* Zip Code file.
6. Create a buffer class to read and write the data file header record
7. Repeat Group Project 1.0 with these two new *length-indicated* files.
8. Create (in RAM), write (as a file), and read (back into RAM), a primary key index that can be used to display the Zip Code data for all Zip Codes listed on the command line.
   * Design and use a header record for the primary key index file (facilitates reading and writing the primary key index more flexibly)
   * Use a command line flag (e.g. **-Z56301**) to indicate each Zip Code record to search for.
   * If the Zip Code record *is not* in the file, display a message to that effect.
   * If the Zip Code record *is* in the file, display the complete Zip Code record on a line with each field labeled.
   * Test Run Demonstration: for each of the two Zip Code data and index file pairs
     1. Run the test program - include searches (on the command line) for several valid Zip Codes and at least one invalid Zip Code.
     2. the program will load the primary key index file into an appropriate container object in RAM
     3. After opening your indexed data file, the only information that should reside in RAM (in appropriate container objects) is:
        1. data file header information
        2. the primary key index (after being read from the primary key index file)
        3. an (unpacked) Zip Code record object
     4. Repeat for versions of the two XLSX files which have their columns re-ordered — there should be no change in the test results.
9. Document (*extensively*) your C++ source code with comments and Doxygen tags.
10. Create a Doxygen PDF of your buffer class and application program code.
11. Create a user guide showing how to use your program (including how to use the command line options, and how the output should appear)
12. Create a design document (a draft of this document should be submitted to the Dropbox by the end of one week)
13. Create a test document

**Architecture**

**Implementation**

We have done the first four requirements on the first project and we will be using the code we have created in the past on this project. These files include the zipCode.cpp, zipCode.h, buffer.h and buffer.cpp files.

5. For the fifth requirement, we will be creating a new buffer file called the LIbuffer. To fit the requirements, we will be making a function called Pack and Unpack as well as a constructor, and a read and write function. Descriptions for these functions will be down below.

6-7. For the sixth requirement, we will be making a new buffer file called DHbuffer with functions like Pack and readHeader to read and write. It is going to be able to write to the header and read from the header. We will be reading to the header so we know how to organize the data. We will also be making a filed called dhObject for various functions. It will take the information from the header and will be put into data members in order to work with it.

8. For part 8, we will be creating primary key indexes associated with the files and we will be using command line options to display records.

9-13. We have done most of the following requirements in a previous project and we will adding tags to better work with Doxygen. We will be creating the user guide and finalize the design document after we have completed the program. After that we will make sure to create a script file and conduct a couple of tests to ensure that the program is working as intended.

Below are all of the files and functions that we are be going to be implementing in this program:

**zipCode**

This file will include functions that will set the values of zip code, placename, state, county, latitude and longitude and as well as functions to get those values after they are set.

class zipCode

**public:**

zipCode() - This is a constructor that sets valid values to zipcode, state, longitude and latitude

const int getZip() – Accessor for placename in file zipcode

const std::string getPlacename() – Accessor for the placename in file zipcode

const std::string getState() – Accessor for the state in file zipcode

const std::string getCounty() – Accessor for the county in file zipcode

const double getLat() – Accessor for the latitude in file zipcode

const double getLon() – Accessor for the longitude in the file zipcode

void setZip(int Zipcode) - Mutator for the zipcode

void setPlacename(std::string Placename) – Mutator for the placename

void setState(std::string State) – Mutator for the state

void setCounty(std::string County) – Mutator for the county

void setLat(double Latitude) – Mutator for the latitude

void setLon(double Longitude) – Mutator for the longitude

**Buffer**

This file includes a class for a buffer which includes methods to read data from a string and unpack that string into a vector of strings.

class DelimTextBuffer

**public:**

DelimTextBuffer(char) – This is the constructor taking in the delimiter character as a parameter

int Read(std::string) – This reads the string that is passed in and sets the buffer equal to the string.

int Unpack(std::vector<std::string>&) – Unpacks the buffer into the vector of strings that is passed in as a parameter. This returns 1 if successful.

std::string getBuffer() – This is the accessor to the buffer field

**LIbuffer**

This header file includes the LIbuffer class. It includes methods to read, write, pack and unpack strings. The class is used for working with length indicated, delimited records. It will be reading from an object, and its going to be unpacking into a length indicated file.

class LIbuffer

**public:**

LIbuffer(char c) – This is the constructor taking in the delimiter character as a parameter

int Pack(zipCode& record) – This packs the fields vector with the data members from a zipCode record.

int Unpack(zipCode& record) – This unpacks the buffer into a zipCode record

int Read(std::ifstream& file) – This reads a length indicated record from an input file stream

int Write(std::fstream& file) – The size of the record and contents(with delimiter character) are written to the file

int getSize() – This is the accessor to the buffer size

void Clear() – This is the clear function and the size is set to 0. The fields are cleared and the buffer is now an empty string.

**DHBuffer**

This header file includes the class DataHeaderBuffer.

class DataHeaderBuffer

**public:**

DataHeaderBuffer – This is the main constructor for this class.

int Pack(dhObject& object) – This is the function used to pack from the header.

int readHeader(std::ifstream& file) – This is the function responsible for reading from the header.

int writeHeader(std::fstream& file) – This is the function responsible for writing to the header.

void setFieldpairs(std::vector<field> pairs) – This function sets the pairs up.

void Clear() – This clears the function.

**dhObject**

This header includes the class dhObject. This includes functions that will set the values of file structure type, header record size, number of bytes for each record size integer, size format type, index file name, record count and count of fields per record and as well as functions to get those values after.

class dhObject

**public:**

dhObject() – This is a constructor to set valid values to the data members of the class

const int getFileversion() – This is the accessor for the file version

const int getHeader\_size() – This is the accessor for the header\_size

const int getRecordcount() – This is the accessor for the record count

const int getFieldcount() – This is the accessor for the field count

const int getPrimekey() – This is the accessor for the prime key

const std::string getFiletype() – This is the accessor for the file type

const std::string getSizeformat() – This is the accessor for the size format

const std::string getIndexfile() – This is the accessor for the index file

const std::vector<std::string> getFieldname() – This is the accessor for the fieldname vector

void setFileversion(int Fileversion) – This is the mutator for the file version

void setHeader\_size(int Headersize) – This is the mutator for the header size

void setRecordcount(int Recordcount) – This is the mutator for the record count

void setFieldcount(int Fieldcount) – This is the mutator for the field count.

void setPrimekey(int Primekey) – This is the mutator for the prime key.

void setFiletype(std::string Filetype) – This is the mutator for the filetype.

void setSizeformat(std::string Sizeformat) – This is the mutator for the size format.

void setIndexfile(std::string Indexfile) – This is the mutator for the index file.

void setFieldname(std::string Fieldname) – This is the mutator for the fieldname.

void setFieldtype(std::string Fieldtype) – This is the mutator for the field type.