

SequenceSetGroupProject

v1

Generated by Doxygen 1.8.17

1 Class Index	2
1.1 Class List	2
2 File Index	2
2.1 File List	2
3 Class Documentation	2
3.1 SequenceSet::Block Struct Reference	2
3.1.1 Detailed Description	2
3.1.2 Member Data Documentation	3
3.2 SequenceSet::Index Struct Reference	3
3.2.1 Detailed Description	3
3.2.2 Member Data Documentation	4
3.3 SequenceSet Class Reference	4
3.3.1 Detailed Description	5
3.3.2 Constructor & Destructor Documentation	5
3.3.3 Member Function Documentation	6
4 File Documentation	12
4.1 main.cpp File Reference	12
4.2 main.cpp	13
4.3 readme.txt File Reference	14
4.4 script_test.txt File Reference	14
4.5 script_main.txt File Reference	14
4.5.1 Variable Documentation	14
4.6 SequenceSet.cpp File Reference	15
4.6.1 Function Documentation	15
4.7 SequenceSet.cpp	16
4.8 SequenceSet.h File Reference	27
4.9 SequenceSet.h	27
4.10 tester.cpp File Reference	29
4.10.1 Function Documentation	29
4.11 tester.cpp	30
4.12 testSequenceSet.cpp File Reference	30
4.12.1 Function Documentation	31
4.13 testSequenceSet.cpp	31
4.14 us_postal_codes_column_reorder.txt File Reference	32
4.15 us_postal_codes_formatted.txt File Reference	32
4.16 us_postal_codes_row_randomized.txt File Reference	32
4.17 us_postal_codes_sequence_set_file.txt File Reference	32
Index	33

1 Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

SequenceSet::Block	2
SequenceSet::Index	3
SequenceSet	4

2 File Index

2.1 File List

Here is a list of all files with brief descriptions:

main.cpp	12
SequenceSet.cpp	15
SequenceSet.h	27
tester.cpp	29
testSequenceSet.cpp	30

3 Class Documentation

3.1 SequenceSet::Block Struct Reference

```
#include <SequenceSet.h>
```

Collaboration diagram for SequenceSet::Block:

Public Attributes

- [Block *](#) [next](#)
- [Block *](#) [previous](#)
- int [records_count](#)
- std::vector< std::string > [data](#)

3.1.1 Detailed Description

Definition at line 100 of file [SequenceSet.h](#).

3.1.2 Member Data Documentation

3.1.2.1 **next** `Block* SequenceSet::Block::next`

Definition at line 101 of file [SequenceSet.h](#).

3.1.2.2 **previous** `Block * SequenceSet::Block::previous`

Definition at line 101 of file [SequenceSet.h](#).

3.1.2.3 **records_count** `int SequenceSet::Block::records_count`

Definition at line 102 of file [SequenceSet.h](#).

3.1.2.4 **data** `std::vector< std::string > SequenceSet::Block::data`

Definition at line 104 of file [SequenceSet.h](#).

The documentation for this struct was generated from the following file:

- [SequenceSet.h](#)

3.2 SequenceSet::Index Struct Reference

```
#include <SequenceSet.h>
```

Collaboration diagram for SequenceSet::Index:

Public Attributes

- `int` [key](#) [4]
- `Block *` [block](#) [4]
- `Index *` [subTree](#) [4]
- `Index *` [nextNode](#)
- `Index *` [parent](#)

3.2.1 Detailed Description

Definition at line 110 of file [SequenceSet.h](#).

3.2.2 Member Data Documentation

3.2.2.1 **key** `int SequenceSet::Index::key[4]`

Definition at line 111 of file [SequenceSet.h](#).

3.2.2.2 **block** `Block* SequenceSet::Index::block[4]`

Definition at line 112 of file [SequenceSet.h](#).

3.2.2.3 **subTree** `Index* SequenceSet::Index::subTree[4]`

Definition at line 113 of file [SequenceSet.h](#).

3.2.2.4 **nextNode** `Index * SequenceSet::Index::nextNode`

Definition at line 113 of file [SequenceSet.h](#).

3.2.2.5 **parent** `Index * SequenceSet::Index::parent`

Definition at line 113 of file [SequenceSet.h](#).

The documentation for this struct was generated from the following file:

- [SequenceSet.h](#)

3.3 SequenceSet Class Reference

```
#include <SequenceSet.h>
```

Collaboration diagram for SequenceSet:

Classes

- struct [Block](#)
- struct [Index](#)

Public Member Functions

- [SequenceSet](#) ()
- [SequenceSet](#) (int b_size, int r_size, float d_cap, std::string i_filename, std::string o_filename)
default constructor.
- [~SequenceSet](#) ()
copy constructor
- void [create](#) ()
destructor
- void [load](#) ()
Method: load param:nreturn:npurpose:here we load blocks from the sequence set file into ram.
- void [close](#) ()
Method: close param:none return:none purpose:close files if needed.
- bool [is_empty](#) (int flag, int block, int record, int field)
- std::vector< int > [search](#) (std::string search_term)
- std::string [get_field_from_record](#) (int field, int record, int block)
- void [populate](#) ()
Method: populate param: return: purpose: we count blocks and records.
- void [insert](#) (std::string new_record)
Method: insert param: return: purpose:
- void [delete_record](#) (int block, int record)
Method: remove param: return: purpose:
- void [update](#) (int block, int record, int field, std::string new_field)
Method: update param:int block, int record, int field, std::string new_field return: updated record purpose:update a record.
- void [display_record](#) (int record, int block)
Method: display_record param:int record, int block return: record purpose:display record in a file.
- void [display_field](#) (int field, int record, int block)
Method: display_field param:int field, int record int block return:field in a record purpose:display fields.
- void [display_file](#) (int limit)
Method: display_file param:int limit return:file purpose:return file.
- void [display_SS](#) ()
Method: display_SS param:n/a return:Sequence set purpose:display sequence set.
- void [validate](#) ()
Method: validate param:nreturn:npurpose:validate a record.
- void [developer_show](#) ()
Method: developer_show param:nreturn:npurpose:
- int [search_file](#) (int primKey)
- std::vector< int > [get_field_range_tuple](#) (int field_index)
- void [nsew_most](#) (std::string state)
- void [state_and_place_from_zip](#) (std::string zip)

3.3.1 Detailed Description

Definition at line 33 of file [SequenceSet.h](#).

3.3.2 Constructor & Destructor Documentation

3.3.2.1 SequenceSet() [1/2] `SequenceSet::SequenceSet ()`

\Here we have the first constructor for the [SequenceSet](#) i think this will be deleted in the end @param int b_size, int r_size @return n/a @purpose this will initialize some of our data and open the file to default

Definition at line 77 of file [SequenceSet.cpp](#).

Here is the call graph for this function:

3.3.2.2 SequenceSet() [2/2] `SequenceSet::SequenceSet (`

```
int b_size,
int r_size,
float d_cap,
std::string i_filename,
std::string o_filename )
```

default constructor.

\Here we have the constructor for the [SequenceSet](#) that takes in all the values relivant to the header and saving @param int b_size, int r_size, int d_cap, std::string i_filename, std::string o_filename @return n/a @purpose this will initialize some of our data and open the file and output file \this is the constructor for the header

Definition at line 98 of file [SequenceSet.cpp](#).

Here is the call graph for this function:

3.3.2.3 ~SequenceSet() `SequenceSet::~~SequenceSet ()`

copy constructor

\Here we have the first destructor for the [SequenceSet](#) @param n/a @return n/a @purpose this free memory

Definition at line 118 of file [SequenceSet.cpp](#).

3.3.3 Member Function Documentation

3.3.3.1 create() `void SequenceSet::create ()`

destructor

Method: create param:none return:none purpose: this will create the empty file with just the header and any data in the data array.

Your header record should include the following components: -sequence set file type -header record size -block size {default to (512B / block)} -maximum count of records per block -minimum capacity: 50% -(for simplicity, require an even number) -record size -count of fields per record -field info triple (tuple) {AoS or SoA} -name or ID -size -type schema -(format to read or write) -indicate field which serves as the primary key -pointer to the block avail-list -pointer to the active sequence set list -block count -record count -stale flag -Simple [Index](#) (10.3) -file name -schema information here i am making the header components to be at the top of the file

here is a design desicion: SoA or AoS here structure

here is an array of structures

write the header

Definition at line 155 of file [SequenceSet.cpp](#).

Here is the call graph for this function:

3.3.3.2 load() `void SequenceSet::load ()`

Method: load param:none return:none purpose:here we load blocks from the sequence set file into ram.

function prototype for [create\(\)](#) that creates empty file for the header any it contains create a local file for loading in that data

if the file ended then tell the user and exit

go through each line of the file

if we find the end of header tag then break

cut it into words. look for "Fields:" Record the fields and stop

here are the strings to find what field is in what spot. a function to strip spaces would be ideal here

here are the store of index's for what in what order

get the line

split it into section

for each one see if it is one of the identifiers above and if so store its location

start a counter

go while we still have lines and are not taking too many fields

if we find the end of header tag then break

take each line which will house the field data

chop it and put it into the correct vector to be used later.

increase since we have another field that was specified

close files

Definition at line [219](#) of file [SequenceSet.cpp](#).

Here is the call graph for this function:

3.3.3.3 close() `void SequenceSet::close ()`

Method: close param:none return:none purpose:close files if needed.

function prototype for [load\(\)](#) that load block of sequence set file into ram

Definition at line [301](#) of file [SequenceSet.cpp](#).

3.3.3.4 is_empty() `bool SequenceSet::is_empty (`
 `int flag,`
 `int block = -1,`
 `int record = -1,`
 `int field = -1)`

function prototype for [close\(\)](#) that is called when file needs to be closed

Method: `is_open` param: `int` flag 0 - file 1 - block 2 - record 3 - field return: `bool` true if empty and false if populated
purpose: to know the state of a structure

if nothing is given but the flag then it will do input/output on command line for user and will take in the index's of requested whatever structure and tell you its status this will check the status of requested

file or the whole linked list

block

record or field

block is valid

Definition at line [334](#) of file [SequenceSet.cpp](#).

3.3.3.5 search() `std::vector< int > SequenceSet::search (`
 `std::string search_term)`

prototype for [is_empty\(\)](#) to know the state of the structure

Method: `search` param: return: purpose:

Definition at line [393](#) of file [SequenceSet.cpp](#).

3.3.3.6 get_field_from_record() `std::string SequenceSet::get_field_from_record (`
 `int field,`
 `int record,`
 `int block)`

function prototype for `search(string)` to search for specific record in the file from user input if we are not in range or acceptable give null

Definition at line [762](#) of file [SequenceSet.cpp](#).

Here is the call graph for this function:

3.3.3.7 populate() `void SequenceSet::populate ()`

Method: populate param: return: purpose: we count blocks and records.

we open the file and skip the header and loop the rest we create an empty node for a btree current record being copied

current block number int primary_key_i;

pointer to current node in_file.open(in_filename);

make btree node as neccessary

make a new node every 3 primary keys

first node skips this

move onto next node

remember where we are

node count is 0

if first then its the root

fill the children of the b tree

make next (or first) [Block](#):

get the empty block

prev is null

next too

resize the array to be the length of the block sizes

resize it for the length of a record //ERROR

increase block count each iteration, and if it isnt 0 like the first iteration then set the first in the sequence set to be b

if not then send it to the next node.

while block isn't ___% full, keep filling:

get the primary key and add it to the tree DO CONTINUE ON FROM HERE

Build the B+ tree up from the "linked list" structure

Definition at line [433](#) of file [SequenceSet.cpp](#).

Here is the call graph for this function:

3.3.3.8 insert() `void SequenceSet::insert (`
`std::string new_record)`

Method: insert param: return: purpose:

function prototype for [populate\(\)](#) that creates an empty node for a btree all blocks filled make a new one

Definition at line [530](#) of file [SequenceSet.cpp](#).

Here is the call graph for this function:

3.3.3.9 delete_record() `void SequenceSet::delete_record (`
`int block = -1,`
`int record = -1)`

Method: remove param: return: purpose:

function prototype for [insert\(strint\)](#) that inserts a new record into the file from user input

Definition at line [597](#) of file [SequenceSet.cpp](#).

3.3.3.10 update() `void SequenceSet::update (`
`int block,`
`int record,`
`int field,`
`std::string new_field)`

Method: update param:int block, int record, int field, std::string new_field return: updated record purpose:update a record.

function prototype for [delete_record\(int, int\)](#) that deletes specific record from user input

Definition at line [630](#) of file [SequenceSet.cpp](#).

Here is the call graph for this function:

3.3.3.11 display_record() `void SequenceSet::display_record (`
`int record = -1,`
`int block = -1)`

Method: display_record param:int record, int block return: record purpose:display record in a file.

function prototype for [update\(int, int, string\)](#) that updates a record, field or adds new field

Definition at line [675](#) of file [SequenceSet.cpp](#).

3.3.3.12 display_field() `void SequenceSet::display_field (`
 `int field = -1,`
 `int record = -1,`
 `int block = -1)`

Method: display_field param:int field, int record int block return:field in a record purpose:display fields.

function prototype [display_record\(int, int\)](#) displays specific record request by user input

Definition at line 714 of file [SequenceSet.cpp](#).

Here is the call graph for this function:

3.3.3.13 display_file() `void SequenceSet::display_file (`
 `int limit = -1)`

Method: display_file param:int limit return:file purpose:return file.

function prototype [display_field\(int, int, int\)](#) displays specific field request by user input

Definition at line 800 of file [SequenceSet.cpp](#).

3.3.3.14 display_SS() `void SequenceSet::display_SS ()`

Method: display_SS param:n/a return:Sequence set purpose:display sequence set.

function prototype [display_file\(int\)](#) displays file request by user input

Definition at line 824 of file [SequenceSet.cpp](#).

3.3.3.15 validate() `void SequenceSet::validate ()`

Method: validate param:nreturn:npurpose:validate a record.

function prototype [display_SS\(\)](#) to display the sequence set

Definition at line 866 of file [SequenceSet.cpp](#).

Here is the call graph for this function:

3.3.3.16 developer_show() `void SequenceSet::developer_show ()`

Method: developer_show param:nreturn:npurpose:

function prototype [validate\(\)](#) to validate a record in the file

Definition at line 899 of file [SequenceSet.cpp](#).

3.3.3.17 search_file() `int SequenceSet::search_file (`
`int primKey)`

function prototype [developer_show\(\)](#) that creates the columns the record will be displayed into

3.3.3.18 get_field_range_tuple() `std::vector< int > SequenceSet::get_field_range_tuple (`
`int field_index)`

function prototype [search_file\(int\)](#) searches for a file

often we want where the characters index is in the record which is stored in the range but extracting that range isn't easy so here's a function to do it

Definition at line [916](#) of file [SequenceSet.cpp](#).

Here is the call graph for this function:

3.3.3.19 nsew_most() `void SequenceSet::nsew_most (`
`std::string state)`

function prototype [get_field_range_tuple\(int\)](#) for extracting the range of character index in a record

Definition at line [968](#) of file [SequenceSet.cpp](#).

Here is the call graph for this function:

3.3.3.20 state_and_place_from_zip() `void SequenceSet::state_and_place_from_zip (`
`std::string zip)`

Definition at line [927](#) of file [SequenceSet.cpp](#).

Here is the call graph for this function:

The documentation for this class was generated from the following files:

- [SequenceSet.h](#)
- [SequenceSet.cpp](#)

4 File Documentation

4.1 main.cpp File Reference

```
#include "SequenceSet.cpp"
Include dependency graph for main.cpp:
```

4.2 main.cpp

```

00001 /*
00002   Authors: Jacob Hopkins, Misky Abshir, and Tyler Willard
00003   Date: 4/27/2020
00004   Due: 5/1/2020
00005
00006   TODO:
00007     - Create SequenceSet Class in the 'SequenceSet.h' file
00008     - Create a test driver for the class above
00009     - Create the program for using the txt files with the class
00010     - Create the design document for
00011         - The Class
00012         - The Test Driver
00013     - A User Manual
00014     - Doxygen the Code
00015
00016   Specifications for a program which uses the Sequence Set class:
00017     The application program will iterate through the sequence set displaying (neatly)
00018     the Northernmost, Southernmost, Easternmost, and Westernmost zip code for a specified state.
00019     {You can verify the results by sorting the Excel source data file first by state, then by
00020     longitude or by latitude}
00021
00022     Also, the application program, using a different set of command line flags,
00023     will display (neatly) the State and Place Name for a specified Zip Code (or set of Zip Codes).
00024
00025     Run the test driver program to build the full Sequence Set file (and index file);
00026     Run the application program specifying the Sequence Set file and the State on the command line;
00027     Use the Unix script command to show:
00028         the building of the Sequence Set file,
00029         the repeated running of the application program and its output for several states,
00030         the repeated running the application program to display the State and Place Name for several
00031         Zip Codes.
00032
00033     In Video 4:
00034     14:30
00035     I don't mind phone call for audi as well, 612-707-2182 that's my cellphone
00036 */
00037
00038
00039 //import header with the SequenceSet class
00040 #include "SequenceSet.cpp"
00041 #include <cstdio>
00042
00043 /*
00044   Here is the main function to start the program.
00045 */
00046 int main(int arg_count, char** arg_values){
00047
00048   /*
00049     Here we declare and initialize the sequence set data. This will call load().
00050   */
00051
00052   SequenceSet data;
00053
00054   data.create();
00055
00056   data.populate();
00057
00058   //while(true)
00059   //  data.display_field();
00060
00061   //data.display_record(); //works most of the time, with the occasional exit
00062
00063   //data.display_file();           //this works great
00064
00065   //data.display_SS();             //this works great
00066
00067   //data.developer_show();         //this works great
00068
00069   /*
00070     std::vector<int> loc = data.search(data.get_field_from_record(0,0,0));
00071     std::cout << "\n" << data.get_field_from_record(0,0,0) << "\nBlock:\t" << std::to_string(loc[0]) <<
00072     "\nRecord:\t" << std::to_string(loc[1]) << "\n";
00073     loc = data.search(data.get_field_from_record(4,1,0));
00074     std::cout << "\n" << data.get_field_from_record(4,1,0) << "\nBlock:\t" << std::to_string(loc[0]) <<
00075     "\nRecord:\t" << std::to_string(loc[1]) << "\n";
00076     loc = data.search(data.get_field_from_record(4,1,1));
00077     std::cout << "\n" << data.get_field_from_record(4,1,1) << "\nBlock:\t" << std::to_string(loc[0]) <<
00078     "\nRecord:\t" << std::to_string(loc[1]) << "\n";
00079     loc = data.search("42.1934"); // from line 28
00080     std::cout << "\n42.1934" << "\nBlock:\t" << std::to_string(loc[0]) << "\nRecord:\t" <<
00081     std::to_string(loc[1]) << "\n";
00082     loc = data.search("yeeet"); // from line 28
00083     std::cout << "\nyeeet" << "\nBlock:\t" << std::to_string(loc[0]) << "\nRecord:\t" <<

```

```

std::to_string(loc[1]) << "\n";
00080  */
00081
00082  //data.insert("");
00083
00084  /*
00085  data.update(0,0,0,"12345");
00086  data.update(0,0,1,"12345");
00087  data.update(0,0,2,"12345");
00088  data.update(0,0,3,"12345");
00089  data.update(0,0,4," 12.345");
00090  data.update(0,0,5,"-12.345");
00091  */
00092
00093
00094  //data.display_SS();
00095  //data.validate();
00096
00097
00098
00099
00100  //std::cout << "You have entered " << arg_count << " arguments:" << "\n";
00101  //for (int i = 0; i < arg_count; ++i)
00102  //    std::cout << arg_values[i] << "\n";
00103
00104  if(*arg_values[1] == 'a'){
00105      std::cout << "Finding the furthest zip codes in: " << arg_values[2] << "\n";
00106      data.nsew_most(arg_values[2]);
00107  }
00108
00109  if(*arg_values[1] == 'b'){
00110      for(int i = 2; i < arg_count; i++){
00111          std::cout << "Finding the State and Place name of zip code: " << arg_values[i] << "\n";
00112          data.state_and_place_from_zip(arg_values[i]);
00113      }
00114  }
00115
00116  //forbidden code here
00117  //wait for character so the screen does not disappear
00118  std::cout << "Press enter...";
00119  getchar();
00120
00121  //return that the program ran correctly
00122  return 0;
00123 }

```

4.3 readme.txt File Reference

4.4 script_test.txt File Reference

4.5 script_main.txt File Reference

Variables

- Script started on

4.5.1 Variable Documentation

4.5.1.1 on Script started on

Definition at line 8 of file [script_main.txt](#).

4.6 SequenceSet.cpp File Reference

```
#include <iostream>
#include <cstdio>
#include <fstream>
#include "SequenceSet.h"
#include <string>
#include <sstream>
```

Include dependency graph for SequenceSet.cpp: This graph shows which files directly or indirectly include this file:

Functions

- `std::vector< std::string > split_string (std::string str, char delimiter)`
Utility Methods.
- `std::vector< char > string_to_vector (std::string s, int n)`
- `std::string add_c_to_a_til_size_of_b (std::string a, std::string b, std::string c, bool front=true)`
- `std::string add_c_to_a_til_size_of_b (std::string a, int b, std::string c, bool front=true)`

4.6.1 Function Documentation

4.6.1.1 [split_string\(\)](#) `std::vector<std::string> split_string (`
 `std::string str,`
 `char delimiter)`

Utility Methods.

param string str, char delimiter return purpose This function will take apart a string and split it by some char delimiter

Definition at line 15 of file [SequenceSet.cpp](#).

4.6.1.2 [string_to_vector\(\)](#) `std::vector<char> string_to_vector (`
 `std::string s,`
 `int n)`

Definition at line 33 of file [SequenceSet.cpp](#).

4.6.1.3 [add_c_to_a_til_size_of_b\(\)](#) [1/2] `std::string add_c_to_a_til_size_of_b (`
 `std::string a,`
 `std::string b,`
 `std::string c,`
 `bool front = true)`

Definition at line 42 of file [SequenceSet.cpp](#).

4.6.1.4 add_c_to_a_til_size_of_b() [2/2] `std::string add_c_to_a_til_size_of_b (`
`std::string a,`
`int b,`
`std::string c,`
`bool front = true)`

Definition at line 55 of file [SequenceSet.cpp](#).

4.7 SequenceSet.cpp

```
00001 #include <iostream>
00002 #include <cstdio>
00003 #include <fstream>
00004 #include "SequenceSet.h"
00005 #include <string>
00006 #include <sstream>
00007
00008
00009
00015 std::vector<std::string> split_string(std::string str, char delimiter){
00016     std::vector<std::string> split_str;
00017     std::string word;
00018
00019     for(char x : str){
00020
00021         if (x == delimiter){
00022             split_str.push_back(word);
00023             word = "";
00024         }else{
00025             word = word + x;
00026         }
00027
00028     }
00029     split_str.push_back(word);
00030     return split_str;
00031 }
00032
00033 std::vector<char> string_to_vector(std::string s, int n){
00034     std::vector<char> v;
00035     v.push_back(n);
00036     for (char c : s){
00037         v.push_back(c);
00038     }
00039     return v;
00040 }
00041
00042 std::string add_c_to_a_til_size_of_b(std::string a, std::string b, std::string c, bool front = true){
00043     int size_to_be = b.size();
00044     std::string new_a = a;
00045     while(new_a.size() < size_to_be){
00046         if(front){
00047             new_a = c + new_a;
00048         }else{
00049             new_a = new_a + c;
00050         }
00051     }
00052     return new_a;
00053 }
00054
00055 std::string add_c_to_a_til_size_of_b(std::string a, int b, std::string c, bool front = true){
00056     std::string new_a = a;
00057     while(new_a.size() < b){
00058         if(front){
00059             new_a = c + new_a;
00060         }else{
00061             new_a = new_a + c;
00062         }
00063     }
00064     return new_a;
00065 }
00066
00067
00068 /*    Class Methods    */
00069
00077 SequenceSet::SequenceSet(){
00078     block_size = 512; //records per block
00079     record_size = 1; //characters per record
00080     in_filename = "us_postal_codes_formatted.txt";
00081     out_filename = "us_postal_codes_sequence_set_file.txt";
00082     default_cap = 0.5;
00083     primary_key_index = 0;
00084     first = NULL;
```

```

00085     end_of_header =
00086         "1234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890";
00087     load();
00088 }
00089
00098 SequenceSet::SequenceSet(int b_size, int r_size, float d_cap, std::string i_filename, std::string
00099     o_filename){
00100     block_size = b_size;
00101     record_size = r_size;
00102     default_cap = d_cap;
00103     in_filename = i_filename;
00104     out_filename = o_filename;
00105     primary_key_index = 0;
00106     first = NULL;
00107     end_of_header =
00108         "1234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890";
00109     load();
00110 }
00111
00118 SequenceSet::~SequenceSet(){
00119     delete(&field_count, &block_size, &record_size, &default_cap, &in_filename, &out_filename, &first,
00120         &root, &primary_key_index);
00121     delete(&end_of_header, &in_file, &out_file, &field_labels, &field_sizes, &field_types);
00122 }
00123
00155 void SequenceSet::create(){
00156     std::string file_type = "ascii";
00157     std::string header_record_size = "22 lines";
00158     record_size = -1;
00159     int max_record_count = -1;
00160     int f_count = field_count;
00161     Block* block_avail = first;
00162     Index* active_list = root;
00163     int block_count = 0;
00164     int record_count = 0;
00165     bool stale = false;
00166
00167     struct field_tuple{
00168         std::string label;
00169         std::string size;
00170         std::string type;
00171     };
00172     field_tuple(std::string a, std::string b, std::string c){
00173         label = a;
00174         size = b;
00175         type = c;
00176     };
00177     std::vector<field_tuple> fields;
00178     for (int i = 0; i < field_count; i++){
00179         fields.push_back(field_tuple(field_labels[i], field_sizes[i], field_types[i]));
00180     }
00181
00182     out_file.open(out_filename);
00183
00184     out_file << "File Type: " << file_type << "\n";
00185     out_file << "Header Size: " << header_record_size << "\n";
00186     out_file << "Block Size: " << block_size << "\n";
00187     out_file << "Maximum Records: " << max_record_count << "\n";
00188     out_file << "Minimum Capacity: " << default_cap << "%\n";
00189     out_file << "Record Size: " << record_size << "\n";
00190     out_file << "Record Field Count: " << field_count << "\n";
00191     for (field_tuple f : fields){
00192         out_file << f.label << '|' << f.size << '|' << f.type << "\n";
00193     }
00194     out_file << "Primary Key: " << field_labels[0] << "\n";
00195     out_file << "Avail Block Pointer: " << block_avail << "\n";
00196     out_file << "Active List: " << active_list << "\n";
00197     out_file << "Block Count: " << block_count << "\n";
00198     out_file << "Record Count: " << record_count << "\n";
00199     out_file << "Stale Flag: " << stale << "\n";
00200     out_file << out_filename << "\n";
00201     out_file << "This file is for loading blocks into a sequence set." << "\n";
00202     out_file << end_of_header << "\n";
00203
00204     close();
00205 }
00206
00219 void SequenceSet::load(){
00220     std::string line = "";
00221     in_file.open(in_filename);
00222
00223     if (in_file.fail()) {

```

```

00226     exit(1);
00227 }
00228
00230 while(std::getline(in_file, line)){
00232     if(!end_of_header.compare(line))
00233         break;
00234
00237     std::vector<std::string> spaceless_line = split_string(line, ' ');
00238     if (!spaceless_line[0].compare("Fields:")){
00239         std::stringstream field_count_string(spaceless_line[1]);
00240         field_count_string >> field_count;
00241         break;
00242     }
00243 }
00244
00246 std::string field_name_identifier = "Field Name ";
00247 std::string column_range_identifier = "    column range ";
00248 std::string type_identifier = "    type ";
00249
00251 int index_of_field_name = -1;
00252 int index_of_collum_size = -1;
00253 int index_of_type = -1;
00254
00256 std::getline(in_file, line);
00258 std::vector<std::string> field_data_positions = split_string(line, '|');
00260 for (int i = 0; i < field_data_positions.size(); i++){
00261     if (!field_data_positions[i].compare(field_name_identifier))
00262         index_of_field_name = i;
00263     if (!field_data_positions[i].compare(column_range_identifier))
00264         index_of_collum_size = i;
00265     if (!field_data_positions[i].compare(type_identifier))
00266         index_of_type = i;
00267 }
00268
00269
00271 int i = 0;
00273 while (std::getline(in_file, line) && i < field_count){
00275     if(!end_of_header.compare(line))
00276         break;
00277
00279     std::vector<std::string> field_data_split = split_string(line, '|');
00281     field_labels.push_back(field_data_split[index_of_field_name]);
00282     field_sizes.push_back(field_data_split[index_of_collum_size]);
00283     field_types.push_back(field_data_split[index_of_type]);
00284
00286     i++;
00287 }
00288
00290 close();
00291 }
00292
00293 void SequenceSet::close(){
00302
00303     Block *b = first;
00304     while( b != NULL){
00305
00306         b = b -> next;
00307     }
00308     delete(b);
00309
00310     if (in_file.is_open()) {
00311         in_file.close();
00312     }
00313     if (out_file.is_open()) {
00314         out_file.close();
00315     }
00316 }
00317 }
00318
00319 bool SequenceSet::is_empty(int flag, int block = -1, int record = -1, int field = -1){
00336     bool status;
00337
00338     if(flag == 0){
00339         status = (first == NULL);
00340     }
00341
00342     if(flag == 1){
00343
00344         if (block == -1){
00345             std::cout << "Index of Block to check: ";
00346             std::cin >> block;
00347         }
00348
00349         Block *b = first;
00350         while(block > 0){

```

```

00351         status = (b==NULL);
00352         b = b -> next;
00353         block--;
00354     }
00355 }
00356 }
00357
00358 if(flag == 2 || flag == 3){
00359     if (block == -1){
00360         std::cout << "Index of Block to check: ";
00361         std::cin >> block;
00362     }
00363
00364     Block *b = first;
00365     while(block > 0){
00366         status = (b==NULL);
00367         b = b -> next;
00368         block--;
00369     }
00370
00371     if(!status){
00372         if (record == -1){
00373             std::cout << "Index of Record to check: ";
00374             std::cin >> record;
00375         }
00376
00377         status = (b -> data[record] == "");
00378     }
00379 }
00380 }
00381
00382 return status;
00383 }
00384
00385
00393 std::vector<int> SequenceSet::search(std::string search_term){
00394     Block *b = first;
00395     std::vector<int> loc;
00396     int block_count = -1, record_count = -1;
00397
00398     while(b != NULL){
00399         block_count++;
00400         record_count = -1;
00401         for(std::string record : b -> data){
00402             record_count++;
00403             if(record_count > b -> records_count){
00404                 break;
00405             }
00406             if(record.size() <= search_term.size()){
00407                 if(record.find(search_term, 0) != std::string::npos){
00408                     loc.push_back(block_count);
00409                     loc.push_back(record_count);
00410                     return loc;
00411                 }
00412             }
00413         }
00414
00415         b = b -> next;
00416     }
00417     loc.push_back(-1);
00418     loc.push_back(-1);
00419     return loc;
00420 }
00421
00422
00433 void SequenceSet::populate(){
00434     int record_number;
00435     int block_count = -1;
00436
00437     int primary_key_int;
00438     std::string primary_key_tmp;
00439     int index_place = -1, node_count = 0;
00440
00441
00442     Block *prev;
00443     Index *current_node = new Index;
00444
00445     std::string line = "";
00446
00447     in_file.open(in_filename);
00448
00449     while(std::getline(in_file, line)){
00450         if(!line.compare(end_of_header)){
00451             break;
00452         }
00453     }
00454 }

```

```

00455 while(!in_file.eof()){
00456     record_number = 0;
00457
00459     if(++index_place%3 == 0){
00460         if(node_count++ > 0){
00461             current_node->nextNode = new Index;
00462             current_node = current_node->nextNode;
00463         }else
00464             root = current_node;
00465
00466         for (int i = 0; i < 4; i++){
00467             current_node -> key[i] = -1;
00468             current_node -> block[i] = NULL;
00469             current_node -> block[i] = NULL;
00470         }
00471         current_node -> nextNode = NULL;
00472         current_node -> parent = NULL;
00473     }
00474
00476     Block *b = new Block;
00477     b -> previous = NULL;
00478     b -> next = NULL;
00479     b -> data.resize(block_size);
00480     for(int i = 0; i < block_size; i++){
00481         b -> data[i] = "";
00482     }
00483
00484     if(++block_count != 0){
00485         b -> previous = prev;
00486         prev -> next = b;
00487     }
00488     else
00489         first = b;
00490     prev = b;
00491
00492
00494     std::string line;
00495     while(record_number < (block_size * default_cap) && !in_file.eof()){
00496         std::getline(in_file, line);
00497         if(line != " "){
00498             //std::cout << record_number << " -" << line << "-\n";
00499             prev -> data[record_number] = add_c_to_a_til_size_of_b(std::to_string(record_number),
std::to_string(block_size), "0") + line;
00500             record_number++;
00501         }
00502     }
00503
00505     std::string tmp = prev->data[record_number];
00506     primary_key_tmp = tmp.substr(0,5);
00507     primary_key_tmp.resize(6);
00508     primary_key_int = atoi(primary_key_tmp.c_str());
00509     current_node -> key[index_place%3] = primary_key_int;
00510     current_node -> block[index_place%3] = prev;
00511
00512     prev -> records_count = record_number;
00513 }
00514 }
00515 delete(prev);
00516
00518
00519 close();
00520 }
00521
00522
00530 void SequenceSet::insert(std::string new_record){
00531     if(new_record == ""){
00532         bool f = true;
00533         std::vector<std::string> constructed_record;
00534         std::vector<int> ranges = {}, ranges_2 = {};
00535         std::string term;
00536         int i = 0;
00537         for(std::string field: field_labels){
00538             term = "";
00539             ranges = get_field_range_tuple(i);
00540             int length = (ranges[1] - (ranges[0]))+1;
00541             if(i >= 1){
00542                 ranges_2 = get_field_range_tuple(i-1);
00543                 if(ranges[0] - ranges_2[1] >= 2){
00544                     constructed_record.push_back(" ");
00545                 }
00546             }
00547             while(term.size() != length){
00548                 std::cout << "Input " << field << ": ";
00549                 std::cin >> term;
00550                 term = add_c_to_a_til_size_of_b(term, length, " ",f);
00551             }
00552             f = false;

```

```

00553     constructed_record.push_back(term);
00554     i++;
00555 }
00556 for(std::string s: constructed_record){
00557     new_record = new_record + s;
00558 }
00559 }
00560
00561 Block *b = first;
00562 bool placed = false;
00563 int block = -1;
00564
00565 while( b != NULL && !placed){
00566     block++;
00567     if(b -> records_count < block_size){
00568         b -> records_count++;
00569         b -> data[b -> records_count - 1] = std::to_string(b -> records_count-1) + " " + new_record;
00570         placed = true;
00571     }
00572     b = b -> next;
00573 }
00574
00575 if(!placed){
00576     block++;
00577     Block *new_b = new Block;
00578     new_b -> previous = b;
00579     b -> next = new_b;
00580     new_b -> records_count++;
00581     new_b -> data[new_b -> records_count - 1] = std::to_string(new_b -> records_count - 1) + " " +
00582     new_record;
00583 }
00584
00585 std::cout << "\nInserted into: \nBlock\t" << block << "\nRecord\t" << b -> records_count - 1 << "\n";
00586 delete(b);
00587 }
00588
00589
00597 void SequenceSet::delete_record(int block = -1, int record = -1){
00598     Block *b = first;
00599     int b_count = 0, r_count;
00600
00601     if(block == -1){
00602         std::cout << "Enter block index: " << std::endl;
00603         std::cin >> block;
00604     }
00605     if(record == -1){
00606         std::cout << "Enter record index: " << std::endl;
00607         std::cin >> record;
00608     }
00609
00610     while( b != NULL && b_count < block){
00611         b_count++;
00612         b = b -> next;
00613     }
00614
00615     if (record > 0 && record < b -> records_count){
00616         b -> data[record] = "";
00617     }
00618
00619     delete(b);
00620 }
00621
00622
00630 void SequenceSet::update(int block, int record, int field, std::string new_field){
00631     Block *b = first;
00632     int b_count = 0;
00633
00634     while(b != NULL && b_count < block){
00635         b -> next;
00636         b_count++;
00637     }
00638     bool front = field == 0;
00639     if(record >= 0 && record < block_size){
00640         if(field >= 0 && field < field_count){
00641             std::string updated = "", current = b -> data[record];
00642
00643             std::vector<int> loc = get_field_range_tuple(field);
00644             int length = (loc[1] - loc[0])+1;
00645             bool added = false;
00646             int count = -2;
00647             for(char c : current){
00648                 if(count < loc[0] || count > loc[1]){
00649                     updated = updated + c;
00650                 }else if(!added){
00651                     added = true;
00652                     if(new_field.size() <= length){
00653                         updated = updated + add_c_to_a_til_size_of_b(new_field, length, " ", front);

```

```

00654         }else{
00655             updated = updated + new_field.substr(0, length);
00656         }
00657     }
00658     count++;
00659 }
00660     std::cout << updated << "\n";
00661     b -> data[record] = updated;
00662 }
00663 }
00664 delete(b);
00665 }
00666
00667 void SequenceSet::display_record(int record = -1, int block = -1){
00676     Block *b = first;
00677     int b_count = 0;
00678
00679     while(block < 0){
00680         std::cout << "\nEnter block index: ";
00681         std::cin >> block;
00682     }
00683     while(record < 0){
00684         std::cout << "\nEnter record index: ";
00685         std::cin >> record;
00686     }
00687
00688     while( b != NULL && b_count < block){
00689         b_count++;
00690         b = b -> next;
00691     }
00692
00693     std::string record_s = "*Record Not Found*";
00694     if(b != NULL){
00695         int size = b->records_count;
00696         if(record >= 0 && record < size){
00697             record_s = b -> data[record];
00698         }else if(record >= 0 && record <= block_size){
00699             record_s = "*Empty Record*";
00700         }
00701     }
00702
00703     std::cout << "\n\'" << record_s << "\' \n";
00704     delete(b);
00705 }
00706
00714 void SequenceSet::display_field(int field = -1, int record = -1, int block = -1){
00715     Block *b = first;
00716     int b_count = 0;
00717
00718     while(block < 0){
00719         std::cout << "\nEnter block index: ";
00720         std::cin >> block;
00721     }
00722     while(record < 0){
00723         std::cout << "\nEnter record index: ";
00724         std::cin >> record;
00725     }
00726     while(field < 0){
00727         std::cout << "\nEnter field index: ";
00728         std::cin >> field;
00729     }
00730
00731     while( b != NULL && b_count < block){
00732         b_count++;
00733         b = b -> next;
00734     }
00735
00736     std::string record_s = "*Record Not Found*";
00737     if(b != NULL){
00738         int size = b->records_count;
00739         if(record >= 0 && record < size){
00740             record_s = b -> data[record];
00741
00742             if(field >= 0 && field < field_count){
00743                 std::vector<int> ranges = get_field_range_tuple(field);
00744                 int length = (ranges[1] - (ranges[0]-1));
00745                 int start = ranges[0]-1 + std::to_string(block_size).size();
00746                 std::string field_s = record_s.substr(start, length);
00747                 //std::cout << "\n\'" << ranges[0] << "-" << ranges[1] << "\' \n";
00748                 std::cout << "\n\'" << field_s << "\' \n";
00749                 return;
00750             }
00751
00752         }else if(record >= 0 && record <= block_size){
00753             record_s = "*Empty Record*";
00754             std::cout << "\n\'" << record_s << "\' \n";

```

```

00755     }
00756 }
00757
00758 std::cout << "\n\'" << record_s << "\' \n";
00759 delete(b);
00760 }
00761
00762 std::string SequenceSet::get_field_from_record(int field, int record, int block){
00763     Block *b = first;
00764     int b_count = 0;
00765
00766     while( b != NULL && b_count < block){
00767         b_count++;
00768         b = b -> next;
00769     }
00770
00771     std::string record_s;
00772     if(b != NULL){
00773         int size = b->records_count;
00774         if(record >= 0 && record < size){
00775             record_s = b -> data[record];
00776
00777             if(field >= 0 && field < field_count){
00778                 std::vector<int> ranges = get_field_range_tuple(field);
00779                 int length = (ranges[1] - (ranges[0]-1));
00780                 int start = ranges[0]-1 + std::to_string(block_size).size();
00781                 std::string field_s = record_s.substr(start, length);
00782                 return field_s;
00783             }
00784
00785             }else if(record >= 0 && record <= block_size){
00786                 return "";
00787             }
00788         }
00789         delete(b);
00790         return NULL;
00791     }
00792
00793 void SequenceSet::display_file(int limit = -1){
00801     if(limit == -1){limit = block_size;}
00802     Block *b = first;
00803     int count = 0;
00804     while( b != NULL && count < limit){
00805         std::cout << "Block " << count << "\n";
00806         std::cout << "Records in Block " << count << ": " << b -> records_count << "\n";
00807         std::cout << "Head of Records: \n" << b -> data[0] << "\n";
00808         int last = b -> records_count - 1;
00809         std::cout << "Tail of Records: \n" << b -> data[last] << "\n\n";
00810         count++;
00811         b = b -> next;
00812     }
00813     delete(b);
00814 }
00815
00816 void SequenceSet::display_SS(){
00825     Block *b = first;
00826     int count = 0;
00827     std::string empty_records_index_string;
00828
00829     std::cout << "\n\nPress enter to see next block...(Ctrl + C to stop)";
00830     getchar();
00831
00832     while( b != NULL){
00833         std::cout << "\nBlock " << count << "\n";
00834         std::cout << "Records in Block " << count << ": " << b -> records_count << "\n";
00835         int r_count = 0;
00836         empty_records_index_string = "[";
00837         for(std::string r : b -> data){
00838             if(r != ""){
00839                 std::cout << "Record " << r_count << ": \" << r << "\"\n";
00840                 r_count++;
00841             }else{
00842                 empty_records_index_string = empty_records_index_string + std::to_string(r_count) + ", ";
00843                 r_count++;
00844             }
00845         }
00846         std::cout << "Empty Records: " << empty_records_index_string << "]\n";
00847         std::cout << "\n\n";
00848
00849         std::cout << "Press enter to see next block...(Ctrl + C to stop)";
00850         getchar();
00851
00852         count++;
00853         b = b -> next;
00854     }
00855     delete(b);

```



```

00856 }
00857
00858
00866 void SequenceSet::validate(){
00867     Block *b = first;
00868     std::vector<int> loc = get_field_range_tuple(0);
00869     int start = 1 + std::to_string(block_size).size();
00870     int length = 1 + ( loc[1] - loc[0] );
00871     bool error = false;
00872     while( b != NULL){
00873         int last = b -> records_count;
00874         for (int i = 0; i < last-1; i++){
00875             int prev = atoi(b -> data[i].substr(start, length).c_str());
00876             int current = atoi(b -> data[i+1].substr(start, length).c_str());
00877             if( prev > current){
00878                 error = true;
00879                 std::cout << "Out Of Order: " << i << "\n";
00880             }
00881         }
00882         b = b -> next;
00883     }
00884     if(!error){
00885         std::cout << "Validated to be: In Order." << "\n";
00886     }
00887     delete(b);
00888 }
00889
00890
00891
00899 void SequenceSet::developer_show(){
00900     std::cout << "field_count:\t" << field_count << "\n";
00901
00902     std::cout << "field_labels|field_sizes|field_types \n";
00903     for (int i = 0; i < field_labels.size(); i++){
00904         std::cout << field_labels[i] << "|" << field_sizes[i] << "|" << field_types[i] << "\n";
00905     }
00906     std::cout << "\n";
00907 }
00908
00916 std::vector<int> SequenceSet::get_field_range_tuple(int field_index){
00917     std::string s = field_sizes[field_index];
00918     std::vector<std::string> sub_s = split_string(s, '-');
00919     int low = atoi(sub_s[0].c_str());
00920     int high = atoi(sub_s[1].c_str());
00921     std::vector<int> r = {low,high};
00922     return r;
00923 }
00924
00925
00926
00927 void SequenceSet::state_and_place_from_zip(std::string zip){
00928     Block *b = first;
00929     std::string rec, zip_s, state_s, place_s;
00930
00931     std::vector<int> loc_zip = get_field_range_tuple(0);
00932     int start_zip = std::to_string(block_size).size() + loc_zip[0] - 1, length_zip = loc_zip[1] - loc_zip[0] + 1;
00933
00934     std::vector<int> loc = get_field_range_tuple(2);
00935     int start_state = std::to_string(block_size).size() + loc[0] - 1, length_state = loc[1] - loc[0] + 1;
00936
00937     std::vector<int> loc_p = get_field_range_tuple(1);
00938     int start_place = std::to_string(block_size).size() + loc_p[0] - 1, length_place = loc_p[1] - loc_p[0] + 1;
00939
00940
00941     while(b != NULL){
00942         std::vector<std::string> records = b -> data;
00943
00944         int stop = b -> records_count - 2;
00945
00946         for (int i = 0; i < stop; i++){
00947             rec = records[i];
00948
00949             zip_s = rec.substr(start_zip, length_zip);
00950
00951             if(zip_s == zip){
00952                 //std::cout << rec << "\n";
00953
00954                 state_s = rec.substr(start_state, length_state);
00955                 place_s = rec.substr(start_place, length_place);
00956
00957                 std::cout << state_s << " " << place_s << "\n";
00958             }
00959         }
00960     }

```

```

00961     b = b -> next;
00962 }
00963
00964 delete(b);
00965 }
00966
00967
00968 void SequenceSet::nsew_most(std::string state){
00969     Block *b = first;
00970     std::string rec, rec_state, lat_s, long_s;
00971     float lat_f, long_f;
00972
00973     std::vector<int> loc_zip = get_field_range_tuple(0);
00974     int start_zip = std::to_string(block_size).size() + loc_zip[0] - 1, length_zip = loc_zip[1] -
loc_zip[0] + 1;
00975
00976     std::vector<int> loc = get_field_range_tuple(2);
00977     int start_state = std::to_string(block_size).size() + loc[0] - 1, length_state = loc[1] - loc[0] +
1;
00978
00979     std::vector<int> loc_lat = get_field_range_tuple(4);
00980     int start_lat = std::to_string(block_size).size() + loc_lat[0] - 1, length_lat = loc_lat[1] -
loc_lat[0] + 1;
00981
00982     std::vector<int> loc_long = get_field_range_tuple(5);
00983     int start_long = std::to_string(block_size).size() + loc_long[0] - 1, length_long = loc_long[1] -
loc_long[0] + 1;
00984
00985     float east_most = 181; //= atof(get_field_from_record(5,0,0).c_str());
00986     float west_most = -181; //= atof(get_field_from_record(5,0,0).c_str());
00987     float north_most = -91; //= atof(get_field_from_record(4,0,0).c_str());
00988     float south_most = 91; //= atof(get_field_from_record(4,0,0).c_str());
00989
00990     std::string zip_east_most;
00991     std::string zip_west_most;
00992     std::string zip_north_most;
00993     std::string zip_south_most;
00994
00995     while(b != NULL){
00996         std::vector<std::string> records = b -> data;
00997
00998         int stop = b -> records_count - 2;
00999
01000         for (int i = 0; i < stop; i++){
01001             rec = records[i];
01002             rec_state = rec.substr(start_state, length_state);
01003
01004             if(rec_state == state){
01005                 //std::cout << rec << "\n";
01006
01007                 lat_s = rec.substr(start_lat, length_lat);
01008                 long_s = rec.substr(start_long, length_long);
01009
01010                 lat_f = atof(lat_s.c_str());
01011                 long_f = atof(long_s.c_str());
01012
01013
01014                 if(lat_f < south_most){
01015                     south_most = lat_f;
01016                     zip_south_most = rec.substr();
01017                 }
01018                 if(lat_f > north_most){
01019                     north_most = lat_f;
01020                     zip_north_most = lat_f;
01021                 }
01022                 if(long_f <= east_most){
01023                     east_most = long_f;
01024                     zip_east_most = long_f;
01025                 }
01026                 if(long_f > west_most){
01027                     west_most = long_f;
01028                     zip_west_most = long_f;
01029                 }
01030
01031             }
01032         }
01033     }
01034
01035     b = b -> next;
01036 }
01037
01038
01039 std::cout << "\n\nNorth-most lat:" << north_most << "\n";
01040 std::cout << "South-most lat:" << south_most << "\n\n";
01041 std::cout << "East-most long:" << east_most << "\n";
01042 std::cout << "West-most long:" << west_most << "\n";
01043 delete(b);

```

```

01044 }
01045
01046
01047
01048 /*
01049
01050
01051     bool found = false;
01052
01053     while(b != NULL && !found){
01054         std::vector<std::string> records = b -> data;
01055
01056         int stop = b -> records_count - 2;
01057
01058         for (int i = 0; i < stop; i++){
01059             rec = records[i];
01060             rec_state = rec.substr(start_state, length_state);
01061
01062             if(rec_state == state){
01063                 lat_s = rec.substr(start_lat,length_lat);
01064                 long_s = rec.substr(start_long,length_long);
01065                 east_most= atof(long_s.c_str());
01066                 west_most= atof(long_s.c_str());
01067                 north_most= atof(lat_s.c_str());
01068                 south_most= atof(lat_s.c_str());
01069             }
01070
01071             if(found){
01072                 i = stop;
01073             }
01074         }
01075     }
01076
01077     b = b -> next;
01078 }
01079
01080 b = first;
01081
01082
01083
01084
01085
01086 void SequenceSet::nsew_most(std::string state){
01087     float east_most = 0.0;
01088     float west_most = 0.0;
01089     float north_most = 0.0;
01090     float south_most = 0.0;
01091
01092     //std::string zipcode_east_most = 0;
01093     //std::string zipcode_west_most = 0;
01094     //std::string zipcode_north_most = 0;
01095     //std::string zipcode_south_most = 0;
01096
01097     Block *copy = first;
01098
01099     std::vector<int> loc = get_field_range_tuple(2);
01100     int start = std::to_string(block_size).size() + loc[0] - 1, length_state = loc[1] - loc[0] + 1;
01101
01102     std::vector<int> loc_zip = get_field_range_tuple(0);
01103     int start_zip = std::to_string(block_size).size() + loc_zip[0] - 1, length_zip = loc_zip[1] - loc_zip[0] + 1;
01104
01105     std::vector<int> loc_lat = get_field_range_tuple(4);
01106     int start_lat = std::to_string(block_size).size() + loc_lat[0] - 1, length_lat = loc_lat[1] - loc_lat[0] + 1;
01107
01108     std::vector<int> loc_long = get_field_range_tuple(5);
01109     int start_long = std::to_string(block_size).size() + loc_long[0] - 1, length_long = loc_long[1] - loc_long[0] + 1;
01110
01111     std::string lat_s, long_s, r, s, zip;
01112
01113     while(copy != NULL){
01114         int i = 0;
01115         float lat_f, long_f;
01116         while(i < copy -> records_count){
01117             r = copy -> data[i];
01118             s = r.substr(start,length_state);
01119             if (s == state){
01120                 std::cout << r << "\n";
01121                 //find max of n, w, e, s
01122                 lat_s = r.substr(start_lat,length_lat);
01123                 long_s = r.substr(start_long,length_long);
01124                 zip = r.substr(start_zip, length_zip);
01125                 lat_f = atof(lat_s.c_str());
01126                 long_f = atof(long_s.c_str());
01127                 s = r.substr(start,length_state);

```

```

01128         std::cout << lat_f << " - " << long_f << "\n";
01129         if(lat_f <= south_most){
01130             south_most = lat_f;
01131             zipcode_south_most = zip;
01132         }
01133         if(lat_f > north_most){
01134             north_most = lat_f;
01135             zipcode_north_most = zip;
01136         }
01137         if(long_f <= east_most){
01138             east_most = long_f;
01139             zipcode_east_most = zip;
01140         }
01141         if(long_f > west_most){
01142             west_most = long_f;
01143             zipcode_west_most = r.substr(start_zip, length_zip);
01144         }
01145     }
01146     i++;
01147 }
01148 copy = copy -> next;
01149 }
01150
01151 std::cout << "\n\nNorth-most lat:" << north_most << "\n";
01152 std::cout << "South-most lat:" << south_most << "\n\n";
01153 std::cout << "East-most long:" << east_most << "\n";
01154 std::cout << "West-most long:" << west_most << "\n";
01155
01156 std::cout << "\n\nNorth-most zip-code:" << zipcode_north_most << "\n";
01157 std::cout << "South-most zip-code:" << zipcode_south_most << "\n\n";
01158 std::cout << "East-most zip-code:" << zipcode_east_most << "\n";
01159 std::cout << "West-most zip-code:" << zipcode_west_most << "\n";
01160
01161 }
01162
01163
01196
01202
01207
01211
01220
01223

```

4.8 SequenceSet.h File Reference

```

#include <iostream>
#include <string>
#include <iterator>
#include <vector>

```

Include dependency graph for SequenceSet.h: This graph shows which files directly or indirectly include this file:

Classes

- class [SequenceSet](#)
- struct [SequenceSet::Block](#)
- struct [SequenceSet::Index](#)

4.9 SequenceSet.h

```

00001 /*
00002     Authors: Jacob Hopkins, Misky Abshir, Tyler Willard
00003     Date: 4/27/2020
00004 */
00005 #include <iostream>
00006 #include <string>
00007 #include <iterator>
00008 #include <vector>
00009
00010 /*
00011     This is a datatype for handling large file in and out of RAM.
00012
00013     We need

```

```

00014     - constructor(s) / destructor (etc. for in-RAM objects)
00015     - create
00016     - open/load (necessary components of an existing SS {i.e. header record & index file into memory)
00017         consider optionally running the validate method
00018     - close
00019     - is_empty (via a flag, can be applied to either file, a block, a record slot in a block, or a
field within a record)
00020     - search (for a record)
00021     - populate (populate the blocked record file from the input data file)
00022         consider populating to 3/4 capacity as a default parameter
00023         (can be changed for testing block merging, splitting, & record redistribution)
00024     - insert (a record)
00025     - delete (a record)
00026     - update (a field of a record)
00027     - display_record
00028     - display_SS (parameterized to display the whole record or a subset of fields)
00029     - validate (is your sequence set ordered by primary key? Can you get to each record via the index
file?)
00030     - (...private helper functions/methods)
00031     - (...debug functions/methods) {consider using a static debug flag for the class}
00032 */
00033 class SequenceSet
00034 {
00035     private:
00036         struct Block;           //see below
00037         struct Index;
00038         Block *first;
00039         Index *root;
00040         int field_count;        //count of fields per record
00041         int block_size;         //records per block
00042         float default_cap;      //where the program will fill blocks to by default
00043         int record_size;        //number of characters per record
00044         int primary_key_index;
00045         std::string end_of_header;
00046         std::fstream in_file;
00047         std::ofstream out_file;
00048         std::string in_filename; //filename for input
00049         std::string out_filename; //filename for output
00050         std::vector<std::string> field_labels; //labels of each field
00051         std::vector<std::string> field_sizes; //sizes of each field
00052         std::vector<std::string> field_types; //type for each field
00053
00054     public:
00055         SequenceSet();
00056         SequenceSet(int b_size, int r_size, float d_cap, std::string i_filename, std::string o_filename);
00057         ~SequenceSet();
00058         void create();
00059         void load();
00060         void close();
00061         bool is_empty(int flag, int block, int record, int field);
00062         std::vector<int> search(std::string search_term);
00063         std::string get_field_from_record(int field, int record, int block);
00064         void populate();
00065         void insert(std::string new_record);
00066         void delete_record(int block, int record);
00067         void update(int block, int record, int field, std::string new_field);
00068         void display_record(int record, int block);
00069         void display_field(int field, int record, int block);
00070         void display_file(int limit);
00071         void display_SS();
00072         void validate();
00073         //void addIndex(int primKey, Block *b); //! function prototype
00074         addIndex(int, Block) that adds an index in a record */ //! function prototype
00075         delIndex(int primKey); //! function prototype
00076         delIndex(int) that removes an index in a record */
00077
00078     void developer_show();
00079     int search_file(int primKey);
00080     std::vector<int> get_field_range_tuple(int field_index);
00081     void nsew_most(std::string state);
00082     void state_and_place_from_zip(std::string zip);
00083 };
00084
00085 /*
00086     Here we create a Block
00087     block size {default to (512B / block)}
00088
00089     Each active block should include the following components:
00090     count of records ( > 0 )
00091     pointers to preceding & succeeding active blocks
00092     set of records ordered by key
00093
00094     Each avail block should include the following components:

```

```

00096     count of records ( == 0 )
00097     pointer to succeeding avail block
00098
00099 */
00100 struct SequenceSet::Block {
00101     Block *next, *previous;
00102     int records_count;
00103
00104     std::vector< std::string > data; //1 dimensional vector holding all records as 1 string
00105 };
00106
00107 /*
00108     This is an index
00109 */
00110 struct SequenceSet::Index {
00111     int key[4];
00112     Block *block[4];
00113     Index *subTree[4], *nextNode, *parent;
00114 };

```

4.10 tester.cpp File Reference

```

#include <iostream>
#include <string>
#include "SequenceSet.cpp"
Include dependency graph for tester.cpp:

```

Functions

- void [menu](#) ()
- int [main](#) ()

4.10.1 Function Documentation

4.10.1.1 [menu\(\)](#) void [menu](#) ()

Definition at line 6 of file [tester.cpp](#).

4.10.1.2 [main\(\)](#) int [main](#) ()

Definition at line 23 of file [tester.cpp](#).

Here is the call graph for this function:

4.11 tester.cpp

```

00001 #include <iostream>
00002 #include <string>
00003 #include "SequenceSet.cpp"
00004 using namespace std;
00005
00006 void menu() {
00007     cout<<"-----Sequence Set Generator-----"«endl«endl;
00008     cout<<"-----"«endl;
00009     cout << "We use lowercase letters to choose an optain" « endl;
00010     cout << "d: calls the delete method." « endl;
00011     cout << "i: calls the insert method." « endl;
00012     cout << "s: displays all the blocks." « endl;
00013     cout << "r: displays the record." « endl;
00014     cout << "f: displays the fields." « endl;
00015     cout << "b: displays the B+ Tree." « endl;
00016     cout << "c: creates the index file." « endl;
00017     cout << "u: calls the update method." « endl;
00018     cout << "m: will display this menu." « endl;
00019     cout << "x: will end this program." « endl;
00020 }
00021 }
00022
00023 int main()
00024 {
00025     SequenceSet s;
00026     char choice = ' ';
00027
00028     s.populate();
00029     menu();
00030     while(choice!='x'){
00031         cout<<"-----"«endl;
00032         cout<<"Enter Choice"«endl;
00033         cin >> choice;
00034         switch(choice){
00035
00036             case 'd':    s.delete_record(),
00037                         s.create();
00038                         break;
00039             case 'm':    menu();
00040
00041                         break;
00042             case 'r':    s.display_record();
00043                         break;
00044             case 'f':    s.display_field();
00045                         break;
00046             case 's':    s.display_SS();
00047                         break;
00048             case 'i':    s.insert(),
00049                         s.create();
00050                         break;
00051             case 'b':    b.print();
00052                         break;
00053             case 'c':    s.create();
00054                         break;
00055             case 'u':    s.update();
00056                         break;
00057             case 'x':    cout << "Terminating program, goodbye!" « endl;
00058         }
00059     }
00060     cout<<"-----"«endl«endl;
00061
00062     return 0;
00063 }
00064 }

```

4.12 testSequenceSet.cpp File Reference

```

#include <iostream>
#include <cstdio>
#include "SequenceSet.h"

```

Include dependency graph for testSequenceSet.cpp:

Functions

- int [main](#) (int arg_count, char **arg_values)

4.12.1 Function Documentation

4.12.1.1 main() `int main (`
`int arg_count,`
`char ** arg_values)`

Definition at line 24 of file [testSequenceSet.cpp](#).

4.13 testSequenceSet.cpp

```

00001 /*
00002     Authors: Jacob Hopkins, Misky Abshir, and Tyler Willard
00003     Date: 4/27/2020
00004
00005     testSequenceSet.cpp
00006     This is a test program.
00007
00008     Sequence set is a class to handle reading data into and out of files and processing with
00009     performance.
00010
00011     This program is to show the functionality of the SequenceSet class found in 'SequenceSet.h'
00012
00013     In video 4:
00014     14:20
00015 */
00016
00017 #include<iostream>
00018 #include <cstdio>
00019 #include "SequenceSet.h"
00020
00021 /*
00022     Here is the main function of the test driver.
00023 */
00024 int main(int arg_count, char** arg_values){
00025
00026     //show of arguments and example using them
00027     std::cout << "You have entered " << arg_count << " arguments:" << "\n";
00028     for (int i = 0; i < arg_count; ++i)
00029         std::cout << arg_values[i] << "\n";
00030
00031
00032     //introduction
00033     std::cout << "This is the test program for the SequenceSet class. " << std::endl;
00034
00035
00036     /*
00037         Here we declare a SequenceSet named test.
00038     */
00039     std::cout << "Declaring SequenceSet: test" << std::endl;
00040     //SequenceSet test;
00041     std::cout << "Declaring SequenceSet complete." << std::endl << std::endl;
00042
00043
00044     /*
00045         Here we initialize the sequence set test.
00046     */
00047     std::cout << "Initalizing test" << std::endl;
00048     //test = SequenceSet();
00049     std::cout << "test initialization complete." << std::endl << std::endl;
00050
00051     /*
00052
00053     */
00054     std::cout << "" << std::endl;
00055
00056     std::cout << " complete." << std::endl << std::endl;
00057
00058
00059
00060     //forbiden code here
00061     //wait for character so the screen does not disappear
00062     getchar();
00063
00064     //return that the program ran correctly
00065     return 0;
00066 }
00067

```


4.14 us_postal_codes_column_reorder.txt File Reference

4.15 us_postal_codes_formatted.txt File Reference

4.16 us_postal_codes_row_randomized.txt File Reference

4.17 us_postal_codes_sequence_set_file.txt File Reference

Index

- ~SequenceSet
 - SequenceSet, [6](#)
- add_c_to_a_til_size_of_b
 - SequenceSet.cpp, [15](#)
- block
 - SequenceSet::Index, [4](#)
- close
 - SequenceSet, [7](#)
- create
 - SequenceSet, [6](#)
- data
 - SequenceSet::Block, [3](#)
- delete_record
 - SequenceSet, [10](#)
- developer_show
 - SequenceSet, [11](#)
- display_field
 - SequenceSet, [10](#)
- display_file
 - SequenceSet, [11](#)
- display_record
 - SequenceSet, [10](#)
- display_SS
 - SequenceSet, [11](#)
- get_field_from_record
 - SequenceSet, [8](#)
- get_field_range_tuple
 - SequenceSet, [12](#)
- insert
 - SequenceSet, [9](#)
- is_empty
 - SequenceSet, [7](#)
- key
 - SequenceSet::Index, [4](#)
- load
 - SequenceSet, [6](#)
- main
 - tester.cpp, [29](#)
 - testSequenceSet.cpp, [31](#)
- main.cpp, [12](#)
- menu
 - tester.cpp, [29](#)
- next
 - SequenceSet::Block, [3](#)
- nextNode
 - SequenceSet::Index, [4](#)
- nsew_most
 - SequenceSet, [12](#)
- on
 - script_main.txt, [14](#)
- parent
 - SequenceSet::Index, [4](#)
- populate
 - SequenceSet, [8](#)
- previous
 - SequenceSet::Block, [3](#)
- readme.txt, [14](#)
- records_count
 - SequenceSet::Block, [3](#)
- script_test.txt, [14](#)
- script_main.txt, [14](#)
 - on, [14](#)
- search
 - SequenceSet, [8](#)
- search_file
 - SequenceSet, [11](#)
- SequenceSet, [4](#)
 - ~SequenceSet, [6](#)
 - close, [7](#)
 - create, [6](#)
 - delete_record, [10](#)
 - developer_show, [11](#)
 - display_field, [10](#)
 - display_file, [11](#)
 - display_record, [10](#)
 - display_SS, [11](#)
 - get_field_from_record, [8](#)
 - get_field_range_tuple, [12](#)
 - insert, [9](#)
 - is_empty, [7](#)
 - load, [6](#)
 - nsew_most, [12](#)
 - populate, [8](#)
 - search, [8](#)
 - search_file, [11](#)
 - SequenceSet, [5, 6](#)
 - state_and_place_from_zip, [12](#)
 - update, [10](#)
 - validate, [11](#)
- SequenceSet.cpp, [15](#)
 - add_c_to_a_til_size_of_b, [15](#)
 - split_string, [15](#)
 - string_to_vector, [15](#)
- SequenceSet.h, [27](#)
- SequenceSet::Block, [2](#)
 - data, [3](#)
 - next, [3](#)
 - previous, [3](#)
 - records_count, [3](#)

- SequenceSet::Index, [3](#)
 - block, [4](#)
 - key, [4](#)
 - nextNode, [4](#)
 - parent, [4](#)
 - subTree, [4](#)
- split_string
 - SequenceSet.cpp, [15](#)
- state_and_place_from_zip
 - SequenceSet, [12](#)
- string_to_vector
 - SequenceSet.cpp, [15](#)
- subTree
 - SequenceSet::Index, [4](#)
- tester.cpp, [29](#)
 - main, [29](#)
 - menu, [29](#)
- testSequenceSet.cpp, [30](#)
 - main, [31](#)
- update
 - SequenceSet, [10](#)
- us_postal_codes_column_reorder.txt, [32](#)
- us_postal_codes_formatted.txt, [32](#)
- us_postal_codes_row_randomized.txt, [32](#)
- us_postal_codes_sequence_set_file.txt, [32](#)
- validate
 - SequenceSet, [11](#)