CS210-001: Assignment 7
Fall 2019 (201930)

Due Date and Time: Thursday, November 28, 2019 at 11:20 AM

(100 marks) In this problem, you are required to write a program to index a file. The index should be implemented using the B-tree algorithms given on the CS210 Algorithms web page. The algorithms and data structures should be implemented as C++ classes.

PART 1

You will index the `censusdata.txt` file in the `/home/venus/hilder/cs210/assignment7/datafiles` directory. This file is a text file containing U.S. census records for 100 adults, where each record contains 15 comma-separated fields. It doesn't really matter what these fields contain as we're really only interested in the first field anyway. The first field is a unique identifier attached to each record and will be used as the key value in your B-tree. You can copy `censusdata.txt` into your home directory, if you wish, or you can save space in your home directory and read it directly from my directory.

PART 2

Prior to doing any programming, you will need to understand some basic concepts of low level file I/O in C++. When your program `open`s a file, a *file descriptor* will be added to your program's *file descriptor table* (a table containing one entry for each open file) to associate your program with the physical file. For simplicity, consider the file descriptor to be a pointer to the physical file indicating the position where the next read from, or write to, the file will occur. This pointer is called the *file marker* and there is always one for each open file in your program. For example, let's assume your program executes the following sequence of statements.

```
string inFileName;
fstream inFile;

inFileName = "censusdata.txt";
inFile.open (inFileName.c_str (), fstream::ate | fstream::in
| fstream::out);
```

This `open` statement `open`s the file named `censusdata.txt` for *both* reading (i.e., `fstream::in`) and writing (i.e., `fstream::out`) and it positions the file marker at the end of the file (i.e., `fstream::ate`). With the file marker positioned at the end of the file, you could write to the file, but a read from the file would result in an end-of-file condition. You could have positioned the file marker at the beginning of the file by omitting `fstream::ate`. However, a write at the beginning of the file would write over any existing data at that position. Consequently, if you want to both read from and write to a file, you need facilities to position the file marker where you want it.

Fortunately, C++ provides facilities for doing this in the form of *seek* and *tell* functions. The `seek` functions reposition the file marker by moving it to a particular position in the file. The `tell` functions report the current position of the file marker. There are actually *three* pairs of `seek` and `tell` functions: one function in each pair for handling input streams and the other for handling output streams. The pairs of `seek` and `tell` functions are described below:

reading

`tellg ()`: Returns the current position of the file marker in an input stream (in all pairs the `g` stands for getting).

writting

`tellp ()`: Returns the current position of the file marker in an output stream (in all pairs the `p` stands for putting).

reading

`seekg (position)`: Reposition the file marker in an input stream.

writing

`seekp (position)`: Reposition the file marker in an output stream.

reading

`seekg (offset, fromPosition)`: Reposition the file marker in an input stream `offset` characters ahead or behind `fromPosition`.

writing

`seekp (offset, fromPosition)`: Reposition the file marker in an output stream `offset` characters ahead or behind `fromPosition`.

Note: `fromPosition` can be any one of `beg` (i.e., seek relative to the beginning of the stream), `cur` (i.e., seek relative to the current position in the stream), or `end` (i.e., seek relative to the end of the stream).

A program that demonstrates the use of these functions can be found in the `fileMarkerDemo.cpp` file in the `/home/venus/hilder/cs210/assignment7/datafiles` directory. Copy this file into your home directory, study it, compile it, and run it until you understand how it works. Some elements of this program will be useful to you for implementing your B-tree program. Note that this program is not "the answer".

PART 3

Consider using the following general algorithm when indexing `censusdata.txt` using your B-tree program:

a. Open the file.

b. Save the offset of the next record to be read.

c. Read the unique identifier (i.e., the key) from the record as an `int`.

d. Insert the key into the B-tree and set the associated offset to point to the location of the record in the file.

e. Skip the rest of the record.

f. Repeat steps b to e until the end-of-file is encountered.

g. Close the file.

PART 4

can we manually edit

Demonstrate that your program works by showing:

a. That you can create B-trees with different branching factors. Show this for `m` = `3` and `16`. So you will need to run your program twice.

b. That you can print the leaf nodes from both B-trees created above in item a. Show both the key and the offset into the file. Print a space character, the vertical line character, and a space character between each node. For example,

```
(153, 0) (176, 65) (275, 124) | (302, 147) (351, 180) |
...
```

c. That you can print the contents of the file in sorted order from both B-trees created above in item a. For example,

how is keys sorted

`(153, 0)` = *contents of the record at position* `0`
`(176, 65)` = *contents of the record at position* `65`
`...`

d. That you can find records from both B-trees created above in item a. Show that you can find the records with the following identifiers:

`1, 141, 262, 4`~~`1`~~`5, 539, 6`~~`2`~~`1, 797, 854, 998`

what identifier

Check find algo missing return statement

If you are working alone, submit to UR Courses: (1) all your source code files (i.e., *only* the `.cpp` and `.h` files) zipped into a single file called `cppandhfiles`, (2) a script file called `part4m3script` showing the compilation and execution of your program for `m` = `3`, and (3) a script file called `part4m16script` showing the compilation and execution of your program for `m` = `16`.

If you are working with a partner, *one of the partners* should submit to UR Courses: (1) a file named `partners` that provides the names and student numbers of the partners and the relative contributions of the partners (should total 100%), (2) all your source code files (i.e., *only* the `.cpp` and `.h` files) zipped into a single file called `cppandhfiles`, (3) a script file called `part4m3script` showing the compilation and execution of your program for `m` = `3`, and (4) a script file called `part4m16script` showing the compilation and execution of your program for `m` = `16`. *The other partner* should submit to UR Courses: (1) a file named `partners` that provides the names and student numbers of the partners and the relative contribution of the partners (should total 100%). Note that *both* partners need to submit the `partners` file.