PROJECT 3: B-TREE

OVERVIEW

This assignment focuses on the implementation of a B-Tree that stores the tree nodes on disk using a random access file for data storage. The B-Tree implementation must support typical operations on tree data structures including look-up, insertion, and deletion of search keys. The B-Tree program must be written as a driver program that accepts input from standard-in to either insert or delete or lookup data in a given B-Tree file. Using the driver program, you must perform an experiment that evaluates the run-time performance of the B-Tree operations.

THE PROGRAM

Write a C program `btree` that implements an algorithm to insert, delete, and lookup information in a B-Tree of degree D, where D is a parameter in the code. The program accepts a file name from the argument list to create or use a file for storing the B-Tree nodes. If the file does not exist, the program must create a new B-Tree file for data storage. If the file exists, it assumes that the provided file is the B-Tree file generated by this program to read and modify data.

Your program must perform the operations on the B-Tree data structure provided in eLearning. In this data structure, search keys are unsigned integers. It must grow or shrink the B-Tree as data is inserted or removed, it must search the B-Tree by traversing the appropriate branches in the tree to locate the tree node that contains the search key. It must handle overflow and underflow situations that may occur as a result of inserting or deleting search keys in the B-Tree data structure. Consistent with the operation of insertion and deletion, it must make the appropriate updates on the B-tree in the random access file so that the B-Tree is consistent after every insertion or deletion operation.

FILE FORMAT FOR B-TREE

The file that stores the B-Tree is a random-access file that allows users to read and write individual nodes in the file. The random-access file must store the root node at the beginning of the file at byte 0 and all other nodes after the root node. Each node has a fixed size so that the byte number where the node starts can be computed given the index of the node. You need to come up with an easy way to store or compute the number of nodes that exist in the tree so that an index for each node can be generated. For example, if the B-Tree contains 50 nodes, the indexes of each node must range between 0 and 50. To simplify reading or writing nodes you may store each node as a byte array in the file.

PROJECT OUTCOMES

- To use random-access files for storing structured data on a secondary storage device.
- To build tree-like data structures on a secondary storage device.
- To use tree-like data structures for performing search operations efficiently.
- To use a Makefile for compiling source code into an executable.

IMPLEMENTATION REQUIREMENTS & SUGGESTIONS

Your program must be started as follows:

```
btree filename [-silent]
```

The parameter `filename` indicates the name of a binary file that contains the data of the B-Tree. The program will prompt the user on an operation to be performed and on the data needed to perform the operation. The optional parameter `-silent` indicates that the program should not print any messages to the screen when prompting for data. This option allows users to run the program on a set of input data written into a file that encodes operation and data on the file.

Some sample code to get started on this project is available in *eLearning*. I would strongly encourage you to review and use this code to develop your solution. Review the slides posted in *eLearning* on trees in general and the B-Tree in particular covering operations on search, insert, and delete in a tree data structure. Also review the slides titled "File IO Slides" posted in *eLearning* under *Project Descriptions* to learn about programming random access files. Sample code for file IO is provided on the slides.

EXPERIMENTATION

Using the completed program run an experiment that uses data from a file to perform operations on the tree. The file data contain information about the operations and the input data to perform them. The file data will be read into memory via redirection of standard-in from the keyboard to the experimental data file. Essentially, the program reads the input as if it was entered via a keyboard. However, the shell will redirect the keyboard input to the file. A test file is given in *eLearning* under *Project Descriptions*. The program processes the user input according to the input data and the following program logic:

1. Read the number of operations that must be peformed.
2. For each operation read on a single line
    a. type as a single character,
    b. operation input data as an integer.

Among the operations, your program must recognize the letters 's' for search, 'i' for insert, and 'd' for delete operations. Your program must ignore any other operations.

As part of your experiment measure the elapsed time from when the program begins the operations to when it completes them. Print out the elapsed time in fractions of seconds and use the real-time clock for time measurements. Use the code provided for a previous project to perform time measurements.

DEMONSTRATION & DELIVERABLES

Submit your complete solution as a single zip file (only standard zip files will be accepted) containing A) source code, B) a single Makefile to compile the program, and C) a README file (if applicable) to the corresponding Dropbox in *eLearning*. In addition, submit the source code and the Makefile as individual files to your own directory on the shared drive under `/shared/dropbox/treichherzer/cop5990`. Be sure to follow the Project Submission Instructions and Code Documentation and Structuring Guidelines posted by your instructor in *eLearning* under Course Materials to avoid unnecessary point deductions on your project grade.

Finally, be sure that your code is well organized. Functions should be refactored so that they can be easily reviewed and maintained. This means that any large function (functions with a body of about 20 or more lines of code) must be broken down into several smaller functions. You should refactor code such as the body of a loop or

an if-else-statement when it exceeds a large number of lines of code or when the statement contains additional nested if-else or loop statements!

DUE DATE

The project is due on the date shown in the syllabus and *eLearning* calendar. Late submissions will not be accepted and the grader will not accept any emailed solutions. The syllabus contains details in regards to late submissions.

IMPORTANT NOTES:

1. Projects will be graded on whether they correctly solve the problem and whether they adhere to good programming practices.

2. Projects must be submitted by the time specified on the due date. Projects submitted after that time will get a grade of zero.

3. Please review UWF's academic conduct policy that was described in the syllabus. Note that viewing another student's solution, whether in whole or in part, is considered academic dishonesty. Also note that submitting code obtained through the Internet or other sources, whether in whole or in part, is considered academic dishonesty. All programs submitted will be reviewed for evidence of academic dishonesty, and all violations will be handled accordingly.

GRADING

This project is worth 100 points in total. The rubric used by the grader is included below. Keep in mind that there will be a substantial deduction if your code does not compile.

Project Submission	Perfect	Deficient		
eLearning	5 points individual files have been uploaded	0 points files are missing		
shared drive	5 points individual files have been uploaded	0 points files are missing		
Compilation	**Perfect**	**Good**	**Attempted**	**Deficient**
Makefile	5 points make file works; includes clean rule	3 points missing clean rule	2 points missing rules; doesn't compile project	0 points make file is missing
compilation	10 points no errors	7 points some warnings	3 points some errors	0 points many errors
Documentation & Program Structure	**Perfect**	**Good**	**Attempted**	**Deficient**
documentation & program structure	5 points follows documentation and code structure guidelines	3 points follows mostly documentation and code structure guidelines; minor deviations	2 points some documentation and/or code structure lacks consistency	0 points missing or insufficient documentation and/or code structure is poor; review sample code and guidelines
Backtracking	**Perfect**	**Good**	**Attempted**	**Deficient**
reads from and writes nodes to a file	10 points correct, completed	7 points minor errors	3 points incomplete	0 points missing
implements deletion of node in the tree	15 points correct, completed	11 points minor errors	4 points incomplete	0 points missing
implements insertion of a node in the tree	15 points correct, completed	11 points minor errors	4 points incomplete	0 points missing

implements search of key elements in the tree	15 points correct, completed	11 points minor errors	4 points incomplete	0 points missing
runs the test program and prints elapsed time to the screen	15 points correct, completed	11 points minor errors	4 points incomplete	0 points missing

I will evaluate your solution as attempted or insufficient if your code does not compile. This means, if you submit your solution according to my instructions, document and structure your code properly, provide a makefile but the submitted code does not compile or crashes immediately you can expect at most 49 out of 100 points. So be sure your code compiles and executes.