**Project 1: Persistent Key-Value Library**

CSC 6712 Distributed Storage Systems

**Overview**

In this project, you’re going to implement a library for storing key-value pairs on disk using a [B-tree](https://en.wikipedia.org/wiki/B-tree) (not a B+-tree). Your library should support the following functionality:

* Creating an empty B-tree
* Checking if a key is in the database and returns its value if so
* Putting a key-value pair in the database (and returning its previous value if there is one)
* Retrieving the next or previous key-value pair for a given key

You do NOT need to implement delete operations.

Your B-tree will have a single type of node that stores keys, values, indices of children, and the index of the parent node. You are not required to implement a B+-tree, which would store all values in the leaves, have separate types of internal and leaf nodes, and connects the leaves into a doubly-linked list. Your tree should not allow duplicate keys. Attempting to put a key-value pair for an existing key will overwrite the value and return the old value.

A diagram of a number

Description automatically generated

**Part I: Library Implementation**

You will need to develop a binary file format. The file format will effectively be an array with each entry taking up 4 KB. The first page in the file should be a 4 KB header that contains only the index of the root node (0 if the tree is empty). Each node will be designed to be stored in exactly 4 KB to match the page size the OS uses for file system access. Keys and values will be exactly 64 bytes each. Rather than storing pointers to memory locations of child nodes, nodes store the page indices.

We will assume that the data may be larger than available RAM. For every search, read the header page to find the root node, read the root node, use it to find the next node, and repeat until the appropriate leaf is found. You should not keep nodes in memory in-between calls to the library API.

When you insert or update a key-value pair, write the modified nodes (pages) back to disk. Once all changed nodes have been written for an operation (e.g., insert or update), call fsync(int fd) to force the OS to write the pages to disk. If a node was read or written previously, it will be returned from the OS disk cache rather than going to disk. Otherwise, if a cache miss occurs, the OS will read the page from the disk.

Your implementation should be tested with automated unit tests. Focus on testing correctness of the API rather than the internal details. For example, if you add 10 key-value pairs, can you read them all back?

**Part II: Benchmarks**

For the normal options, you will perform a different set of benchmarks. You should evaluate two questions:

1) Are read and write performance different? Stated another way, is the B-tree more efficient for one or the other?

For this benchmark, you should measure the total time to read or write different numbers of key-value pairs (e.g., 100 to 1 million increasing in multiples of 10). Repeat 3 times for each number of keys to assess variability in the timings. Create a plot with the number of pairs on the x-axis (log scale) and the elapsed time on the y-axis (also log scale). You can use a [Seaborn point plot](https://seaborn.pydata.org/generated/seaborn.pointplot.html) to show the average and standard deviations for each number of pairs. Use one line for reading and another line for writing.

2) How does OS disk caching impact read performance?

The OS caches pages in RAM, which reduces the number of disk operations needed for reads. You should compare the read performance as more and more of the B-tree is cached in memory. To do this, you will need to start with an empty OS disk cache and record the time needed for reading individual key-value pairs in a sequence.

While Linux makes it easy to clear the OS disk cache, other operating systems do not. To work around this issue, you will create your database file, restart, and then do your read tests. To run multiple trials, you can create multiple database files. (Even if you are using Windows Subsystem for Linux or a virtual machine, you should reboot your host OS. The host OS will cache pages of the image file accessed by the virtual machine.)

To perform this experiment, read 10 randomly chosen keys from each database file, recording the operation sequence number (first, second, etc.) and the run time for reading each key-value pair. Generate a plot in which the x-axis is the operation sequence number, and the y-axis is the run time. If you perform multiple trials (which you should), use a Seaborn point plot to plot the average and standard deviations.

In your report, describe the impact of caching on reads and writes and determine whether your implementation is optimized for reads, writes, or both.

**Part III: Written Report**

You will need to write a 3-4 page report that includes:

* A description of your implementation including how you organize the data and explanations of the operations for finding, inserting, and updating pairs. Use tables, visuals, etc. as needed.
* Your benchmark results (as graphs) for the run times and your interpretations of their meanings. Infer whether your implementation is optimized for one of reads or writes or provides equivalent performance for them. You should also describe the impact of the page cache on read performance.
* Assume that the computer could crash at any point during a write operation. Describe the impact (e.g., data loss, file left in an inconsistent state) of failures at various points.

**Hints**

* I drew out the steps for starting with an empty B-tree and subsequently adding more and more items to make sure I understood the algorithms for inserting values and splitting nodes pretty well before I started coding. I used a B-tree with 3 values and 4 children.
* Next, I implemented an in-memory B-tree before moving to the on-disk B-tree to better understand the algorithms and avoid trying to solve multiple problems at once. My initial implementation only supporting adding and updating key-value pairs in a tree with a single leaf node. I wrote a series of unit tests to validate functionality. I then added support for storing additional pairs by splitting the leaf into two leaves and an internal node used as the new root. After that, I generalized the implementation to work on a tree of any depth.
* You can inspect the binary file using "hexdump -C" or a hex editor.
* I implemented my solution using the [D programming language](https://dlang.org/). Like C, D allows the definition of structs that can be cast to/from byte arrays, making it easy to serialize or deserialize nodes. C, C++, etc. tend to align types to multiples of 4 bytes. I found it useful to write tests for the sizes of the structs using sizeof().  
    
  In Python, you can use the [struct](https://docs.python.org/3/library/struct.html) module to convert between [bytes](https://docs.python.org/3/library/stdtypes.html#bytes) objects and tuples.  
    
  For language such as Java, you could write a wrapper class for the nodes. The constructor can take a 4 KB byte array read from the file and provide getters and setters for the various values that handle serialization / deserialization of values. The class could have a method for returning the underlying byte array for writing back to a file.

**Submission Instructions**

Your source changes should be tracked using GitHub. Create a private GitHub repository for the semester. Add your instructor (rnowling on GitHub) as a member of the repository with read access. Your repository should have a README.md with instructions for building and running your code. Your code should be documented. It would be great if your benchmarks were automated and reproducible by calling a script in the repository. For each project, create a branch capturing the state of the code after its completion (so I can easily see the code at a particular point in time).

Submit a link to the repository and your report (in PDF format) on Canvas.

**Rubric**

|  |  |
| --- | --- |
| **GitHub Repository:** Has a README.md file with instructions sufficient for building and running the code. | 5% |
| **B-tree Implementation:** Implementation is functionally correct with the expected run-time complexity. Functionally is validated through unit tests. B-tree is stored persistently on disk. Nodes are exactly 4 kbs. An appropriate file format was designed. | 30% |
| **Automated Unit Tests:** Some basic automated unit tests are provided and pass. | 5% |
| **Written Report (Overall):** Report is written in a professional manner using proper grammar and spelling. Report is a useful standalone document that can be shared with another student. Report contains a detailed description of the implementation, experimental designs for benchmarks, benchmark results, and interpretations of the results. | 10% |
| **Plots:** Appropriate types of plots were chosen for each analysis. Axes are properly labeled. Used legends if appropriate. Chose appropriate axis limits to make plots readable and avoid misleading interpretations. Font sizes are legible. Figures are saved at high resolutions. | 5% |
| **Data Structure / Algorithm Descriptions:** The data structure design is clearly documented (including the page layouts) and explained. The search, insert/update operations, and search for next/previous key operations are explained clearly. | 15% |
| **Failure Scenarios:** Various failure scenarios are identified and explained. | 15% |
| **Benchmarks:** Experimental designs are correct. Benchmark results are interpreted correctly. | 15% |