1 Introduction

You are to write a library to manage a durable data structure. The durability property means that the data structure will survive system failure without corruption. The code for the above should be in a library, libshadowtree.a.

The data structure SearchTree is a templated class structure with arguments Key and Elmt. The searchTree class implements a search tree with the following operations:

- `bool create(char *name)`: create a new search tree, returns true if successful.
- `bool open(char *name)`: open an existing search tree, returns true if successful. Must lock and perform recovery if necessary.
- `bool insert(Key k, Elmt e)`: Insert an elmt in the tree, returns true if k does not exist in the tree and the element is successfully inserted.
- `bool insert2(Key k1, Elmt e1, Key k2, Elmt e2)`: Insert two elmts atomically in the tree, returns true if k1 and k2 do not exist in the tree and the elements are successfully inserted.
- `bool remove(Key k)`: remove k from the tree, return true if found.
- `Elmt find(Key k, int u=0)`: Find the element corresponding to k that existed update instructions ago, if it exists, otherwise throw an exception.
- `void print(int u=0)`: print the whole tree, u steps ago.

The application is to consist of multiple processes performing operations against a shared database. Note that there does not exist a single database process: rather the database functions are spread uniformly through the various user processes.

ACID Properties The above operations must fulfill ACID properties, that is they must be Atomic (All or nothing), they must be Consistent (Map the database from one well known state to the next), they must provide Isolation (as if one operation is done at a time) and they must be Durable (a successfully completed operation is permanent, even if the computer fails the instant after it completes).

Files You will use a constant number of files.

Locking Locking will be performed using file locks. The database operations must be serializable, that is, occur as if performed one at a time without interleaving.

Shadow Paging Shadow paging is a tree-based technique in which the store as a root node. All updates for this project logically consist of changes to a single node: under shadow paging they physically consist of replacing the path from root to the node change.

For example, in the Figure 1 the result of inserting a node is shown in before and after diagram.

You may assume that writes of 4kb (aligned) are atomic.
Figure 1: Before and after picture of inserting new node into shadow tree. Dotted components in the after picture are not logically part of the new tree but are still physically present.

**Time travel** You will keep the shadow trees with the last 100 update operations. Using `find`, you will be able to look at previous versions of the tree.

**Garbage collection** You must reclaim storage which is no longer needed. Of course, this must be done in a way which is fault tolerant.

## 2 Testing

You should test your program extensively. These tests should include:

- Uniprocess functionality tests
- Uniprocess error returns
- Recovery of damaged stores
- Concurrency testing

To aid us in determining the correctness of your code, you will write a program `print`. Print will print out the contents of the tree in human readable format.

## 3 Instructions for handing in your program

Obviously, your program listing and program turnin must be complete by 2:00 on the due date. **No late programs will be accepted.** Note that this program is considerably larger than the first one, so start on it right away.
Please take particular care to show in your documentation:

- Why your program does not deadlock.
- Why your storage reclamation works.
- How much concurrency can be obtained.
- What you tested.
- What you did not test.
- What works.
- What does not work.

4 Grading Criteria

- Code quality (10 pts)
- Documentation (20 pts)
- Testing (20 pts)
- Correctness (40 pts)
- Concurrency (10 pts)