Concurrent Data Structures
Lab Assignment 1

Parosh Aziz Abdulla
Sarbojit Das

November 20, 2022

Deadline: 2022–12–05.
Please, submit your programs and instructions to run them.

Task 1 Set Abstract Data Type We know that *Abstract Data Types (ADTs)* are mathematical objects that allow us to specify the expected behaviours of implementations of common data structures such as sets, queues, and stacks. In this course's lab assignments, we will simulate ADTs by simple programs that allow atomic operation executions.
Write a program which implements the `SetLib` Abstract Data Type.

- A state of the ADT is a set from the library. (std::set for C++, python set).
- Implement three C++ methods/functions:
  1. `add(elem, out)`: If `elem` exists, return `!out` (`!` means negation) . Otherwise, add `elem` to the set and return `out`.
  2. `rmv(elem, out)`: If `elem` does not exist, return `!out`, otherwise remove it from the set and return `out`.
  3. `ctn(elem, out)`: If `elem` does not exist, return `!out`, otherwise return `out`.

An operation consists of a method name with input and output values. From a given state s, an operation is allowed if a corresponding rule is enabled. For instance, assume that the set $s = \{3, 7, 9\}$. The program allows the operations $(\texttt{add}, 5, \texttt{true})$ and $(\texttt{add}, 7, \texttt{false})$ from s, but not $(\texttt{add}, 8, \texttt{false})$ Test your *ADT* by creating an empty set. Create a test case which is a sequence of 20 operations (includes `add`, `rmv`, and `ctn` randomly). In each

iteration, the test program reads one operation from the sequence and calls one of the functions `add()`, `rmv()`, and `ctn()` to check if it is allowed in the current state. If an operation is not allowed, print the state and the operation that raised the error.

Task 2 Sequential Program Write a program which implements the `SetList` Abstract Data Type using linked list similar to the lecture. That means the state of the ADT is a stored in a linked list. Implement the `add()`, `rmv()`, `ctn()` methods (different from Task 1) as below:

1. `add(elem)`: If `elem` exists, return `false`. Otherwise, add `elem` to the set and return `true`.
2. `rmv(elem, out)`: If `elem` does not exist, return `false`, otherwise remove it from the set and return `true`.
3. `ctn(elem, out)`: If `elem` does not exist, return `false`, otherwise return `true`.

Check whether `SetList` is working exactly like `SetLib`. Test your implementation by creating an empty `SetList`. The program takes a sequence of pairs(method name and input value), and generates a sequence of operations(method name, input, and output value). For instance, the program reads a pair (`rmv`, `2`) from the sequence of pairs, runs the method `rmv(2)` on `SetList`, which gives `true` as output. Then put (`rmv`, `2`, `true`) into the sequence of operations. After getting the sequence of operations, create an empty `SetLib`. For each operation in the sequence, check if it is allowed in the current state of `SetLib`. If there is an error, print the state and the operation. Create a test case that is a sequence of 100 pairs (includes `add`, `rmv`, and `ctn` randomly) and run your test program.

Task 3 Naive Concurrent Program Create an empty `SetList` and an empty sequence of operations. Spawn $N = 2, 4$, and 8 worker threads that run concurrently. Each worker thread takes the same test case, which is a sequence of pairs (method name and input value). For each pair, it runs the respective method on `SetList` with the input and forms an operation (method name, input, and output value). Then it puts the operation into the shared sequence of operations. Create an empty `SetLib`. Read each operation from the shared sequence of operations and checks if the operation is allowed in the current state of `SetLib`. If there is an error, it should print the name of the thread and the operation that raised the error. Create five

test cases of 100 pairs (includes `add`, `rmv`, and `ctn` randomly) and run your test program.

Task 4 Coarse-Grained Synchronization Redo the previous task, but use **Coarse-Grained** synchronization for the worker threads. Implement this *ADT* `CoarseSet` in a separate file. Find linearization points and put the operations to the shared sequence of operations at linearization point. Create and run test 5 cases of 100 pairs(includes `add`, `rmv`, and `ctn`) for worker threads like Task 3.