Definitions

The Program is the design and implementation of a concurrency simulator that models the execution of simultaneous multiple programs on a single-processor system. Each program is made up of a series of statements to be executed, taken from user input. Priority to which program will execute the next statement or series of statements depends on a number of factors.

A program in this assignment is an object that contains a link listed queue of statements to be executed. The Program will have an **active queue** of programs that is actively executing statements, and a second queue, called the **blocked queue**, in which programs are placed if they are suspended due to a lock. Each program has a **shared storage** for 26 possible variables which can be assigned, reassigned and output to user by a program's statements. All variables are an integral type and are initialized to be '0'.

Linked Queue: A queue of link listed nodes will hold both programs and statements, as there are no limits to the number that can be put into either queue.

Time Quantum: Each program has an identical time quantum. This is a integral value that represents the amount of time each program can run before control is transferred to the next program in the active queue.

A statement is a single line of input from the user that is queued up inside a program based on order of input (first read, first in, first out). A statement represents one of five possible actions:

1. **Assignment**: *variable = constant;* This will change one of the 26 possible variables accessible to all programs to the constant from user input. Each variable is shared among all programs and is represented by lower-case letters 'a' to 'z'. Each variable may be assigned an unsigned int up to a value of 100.
2. **Output**: *print variable;* This will output the current value stored in the corresponding variable. Output will be in the format: "program number: value " with a newline between each output.
3. **Begin mutual exclusion***: lock;* Once a program has put a **lock** into place, any subsequent program attempting to execute a lock will be removed from the active queue, and placed at the back of the blocked queue, suspended until an **unlock** statement is issued. If only a single program contains a lock and unlock statement, all programs will run and no program will be suspended.
4. **End mutual exclusion**: *unlock;* This statement ends the lock that was in place during execution. Once this statement is executed, the program at the front of the blocked

queue is immediately removed and placed at the end of the active queue, with the lock that program originally planned to execute as the next statement to be carried out.

5. **Stop execution**: *end;* This signals the program is at an end, meaning all the programs statements have been executed. At this point, the program is removed from the ready queue and not requeued.

Statement Execution Time: Each type of statement requires a specific amount of time to execute based on input from user. The execution time is a positive integral value. The program's time quantum determines the number of statements that can execute before the next program begins its execution. For example, if the program time quantum is 3 and each type of statement has an execution time of 1, a program may execute 3 statements before control moves to the next program.

Purpose

The purpose of this assignment is to simulate how different programs execute their instructions based on a fixed time quantum for each program to operate, and fixed execution times for the 5 statement types.

User input is read in from cin to determine how many programs will be present, and the statements for each program. Each set of program statements terminates with an "end" command, signifying end of instructions for that program. The correct number of programs are instantiated and placed in the ready queue for execution. Then, each statement is read in from cin and placed in the program which was created to hold it. When the "end" command is reached, the next program in the ready queue begins queuing its statements from cin until all input is read. There will be **no preset limit** to the number of programs or statements for each program.

While all these programs and statements are being allocated, dynamic memory allocation is checked to make sure all memory is allocated properly, and any statement or program that fails to allocate will not be queued. A bad_alloc exception is automatically thrown by the compiler, in addition to error handling done by the program itself.

After all programs and statements are queued correctly, execution begins based on the provided time quantum. Each program executes statements for the quantum provided, based on the execution times of each statement. Per the assignment sheet and Stiber in class, if a statement begins execution while the time quantum for that program is not yet 0, that

statement will be allowed to finish execution, regardless of its execution time. For example, if the time quantum is 2 but assignment takes 3 times until to execute, an assignment statement will still execute all the way through when encountered, even though its time is larger than the overall time quantum.

If one program executes a lock statement, in subsequent lock attempting to execute while the first lock is active will be **removed from the active queue and placed at the end of the blocked queue.** Once an unlock statement is executed, the program in first position in the blocked queue will be removed and added to the end of the active queue.

Assignments on the programs' shared storage will execute with no output to user, and the only output will be when a "print variable" statement is executed, which prints to cout in the format: "variable: value " with a newline between each statement.

Per Stiber in class, the input will have no lock that does not have a corresponding unlock, which would leave a program in the blocked queue permanently.

Once all statements from a program have been executed, meaning the "end" statement was encountered, that program is removed from the active queue and not enqueued anywhere else. When all programs have reached their "end," the simulation is complete.

Input

The program reads user input from **cin**. It assumes as per Stiber, no lock will be present without a corresponding unlock, and no unlock will be issued without a previous lock statement. The first line of input will be a series of seven integral values separated by white space as follows:

3 1 1 2 3 1 4

Each number in the order presented represents:

1. The number of programs to be present (3 in this case).
2. The time required for the assignment statement to execute (1).
3. Time required for output to execute (1).
4. Time required for lock to execute (2).
5. Time for unlock to execute (3).
6. Time for end to execute (1).
7. The time quantum for all programs (4).

The remainder of the input will be each statement for an individual program to put a program's queue until all input is read through. Once all input is read, the correct number of programs should be instantiated, each with their own queue list of statements to execute.

Output

Output is only generated when a "print variable" statement from a program is executed, which is sent to **cout.** All output will be in the format:

Program number: variable value (\n)

Program number will be the identifier assigned to the program, "1" for the first, "2" for the second and so on. The value will be an unsigned integer up to 100. If input contains no print variable statements, no output is produced.

Error Handling

The program handles errors in the following situations:

- If the “new” command to allocate dynamic memory fails, the compiler automatically throws a bad_alloc exception, and the method allocating memory returns a **false bool value**, indicating allocation was not successful.
- If the method to get an item from the linked queue attempts to access an item from either an empty list, or the position of the item provided by user is greater than the number of items in that list, an **out_of_range exception** is thrown.
- When clearing a linked queue of all nodes, **a false bool value** is returned by the clearing method if one or more of the Nodes fails to be deleted.
- When removing an item from a linked list based on user providing an item, a **false bool value** is returned by the method if the Node containing the item to be removed is not found in the linked list.
- If an attempt to dequeue an object from an empty queue is made, a **false bool value** is returned indicating that no item could be dequeued.
- If attempting to “peek” at the first item in a queue while the queue is empty, an **out_of_range exception** is thrown.
- If user is trying to get the variable identifier or the value of the variable from a statement that has none, specifically a lock, unlock or end statement, a **logic_error exception** is thrown.
- If an unlock statement is attempting to execute while no lock statement has executed, the unlock statement is skipped over and the next statement is executed instead
- If a program has no statements, meaning it is empty, it is dequeued when the program attempts to execute a statement