KUTZTOWN UNIVERSITY Department of COMPUTER SCIENCE

Invariants, Preconditions, and Postconditions

Proofs of correctness for functions/methods are based upon preconditions, postconditions, and invariants. Particularly in courses such as Programming Languages, your instructor might ask for these within function documentation. They are defined as follows:

- Invariant - a condition that is supposed to be true all the time (except during brief, well-defined intervals).
- Precondition – a condition that must be true before a function (or code segment) executes, in order for the code to work correctly.
- Postcondition - a condition that will be true after a functions returns or at the end of a code segment.

An ordered list is a good example of an ADT with which invariants, preconditions, and postconditions can be demonstrated. An invariant of an ordered list is that both before and after any operation, the list is ordered. This may seem obvious, but work is required to maintain the list order during insertion and removal. And, depending on implementation, the list may not be ordered at some point during an operation.

Consider insertion into an ordered list implemented using a C++ array. If insertion involves appending the new item and then sorting the list, then during the insertion operation the list will not be ordered. This does not violate the invariant, so long as the invariant is true when the operation concludes.

There is no precondition for this operation, as there is no specific requirement for putting an element into a list (assuming no type conflicts). The postcondition for the insertion operation is trivially that the list is ordered when insertion is completed.

An operation with a non-trivial precondition would be removal from the list. Before such an operation can be implemented, it must be determined whether the item to be removed is assumed to be in the list. For a linked list data structure, if it can be assumed that the item is in the list, then there is no need to consider the case that the pointer used for the traversal can become null. This further assumes the user will confirm the presence of an element to be removed before actually calling the remove operation.

Stack and queue operations have non-trivial pre and post conditions. Their invariants are based upon their foundational premises, LIFO for a stack and FIFO for a queue. The user of implementations of these ADTs is expected to check for full before pushing or inserting (if the data structure has limited capacity) and to check for empty before removing, or obtaining a copy of the element on the top or front, respectively, of the construct.

Observing pre and post conditions is critical to crash-free development. For example, if a function that opens a file takes the file stream as a parameter and returns Boolean to indicate whether the file was opened, it will have a postcondition that if it returns true the stream will be initialized or if it returns false the stream will be undefined. Proper usage requires that the user observe the postcondition and check the function’s return value before accessing the stream.