Tutorial 06 — Arrays and Linked Lists

lundi, juin 6

An algorithm to “Search by Index” in a linked list:

Precondition:

The global data accessible by this method is a doubly linked list. An integer `index` is given as input.

Postcondition:

If `index` is negative, or greater than or equal to the `length` of this linked list, then an `IndexOutOfBounds` exception is thrown. Otherwise, the value of the node in this linked list, with the input
`index`, is returned as output.

```
E get (int index) {
    if ((index < 0) or (index ≥ this.length)) {
        throw new IndexOutOfBoundsException();
    } else {
        integer i = 0;
        node currentNode = this.head;
        // Loop Invariant:
        // a) This (that is, the global data being accessed) is a doubly linked list.
        // b) index is an integer input such that 0 ≤ index ≤ this.length - 1.
        // c) i is an integer variable such that 0 ≤ i ≤ index.
        // d) currentNode is the node at position i in this linked list.
        // Bound Function: index - i.
        while (i < index) {
            currentNode = currentNode.next();
            i = i + 1;
        }
    return currentNode.value();
}
```

1

Show that the "loop invariant" shown for the `while` loop in this algorithm really is one.

Proof

An assertion A is a **loop invariant** for this `while` loop if it satisfies all of the following, whenever the algorithm is executed with the problem's precondition satisfied:

- an execution of the loop test `t` has no side-effects — that is, it does not change the value of any inputs, variables, or global data,
- A is satisfied whenever the loop is reached, during an execution of the algorithm starting with the problem's precondition being satisfied, and
- if A is satisfied at the beginning of any execution of the loop body `S` (when the problem's precondition was satisfied when execution of the algorithm started) then A is satisfied, once again, when this execution of the loop body ends

By examination of the code above **before the `while` loop is reached**, we can see the following:

a) `this` is a doubly linked list, as specified in the precondition, and this global data is not modified in any way when the `while` loop is reached, so this proves the first part of the loop invariant

b) Since the `if` statement on line 2 has to fail in order to reach the `while` loop on line 13, we know that the value of `index` is equal to or in between 0 and `this.length - 1`. This establishes the second part of the loop invariant

c) `i` is initialized to 0 on line 5 of the algorithm above. We also know that `index` is greater than 0 due to the test on line 2 failing. Therefore, the third part of the proposed loop invariant is also true.

d) `currentNode` is initialized to the `head` (the beginning) of the linked list, before the `while` loop is reached. The value of `i` is initialized to 0 before the `while` loop executes. This value is incremented inside the body of the loop, along with `currentNode` being set to the next node in the list. This establishes the last part of the loop invariant which states that `currentNode` is the node at index `i` in the list

Now we will examine the loop invariant again, now at the **end of the execution of the `while` loop**.

a) `this` is still a doubly linked list when the end of `while` loop is reached, and this fact is not modified by the steps inside the body of the loop, establishing the first part of the loop invariant.

b) The value of `index` is not changed either, and neither is the `length` of the linked list. So this condition is still satisfied at the end of the loop, as it was at the beginning

c) We know the value of `i` was incremented inside the body of the loop, and at the end of the loop its value is still less than `index`, which establishes the third part of the loop invariant

d) We know that `currentNode` was set to `currentNode.next()` inside the body of the loop, in accordance with `i` being incremented. Therefore, this part of the loop invariant is still true at the end of the `while` loop as well

2

Use this to show that this algorithm is partially correct.

To prove this, it is sufficient to inspect the code to check that the algorithm's execution has no undocumented side-effects. In other words, we have to ensure that no additional inputs or global data are accessed, and no additional data (including inputs) are modified.

- Line 2 contains an `if` statement, checking for the value of `index`. If the test passes, this means its value is less than 0 or greater than the `length` of the linked list, which leads to the exception on line 3 being thrown. The algorithm then terminates, with the postcondition satisfied.
    - Otherwise, if the test fails, we know the value of `index` to be 0 $\leq$ `index` $\leq$ `this.length - 1` and the execution of the algorithm continues
- Lines 5 and 6 do not modify any input or global data, and the `while` loop is reached
- Lines 13-16 contain a `while` loop which, in order for the algorithm to be considered partially correct, must terminate. This means the loop invariant has been satisfied at this point as described above
- Line 17 then contains a `return` statement, and reaching this line satisifes the postcondition for this algorithm, proving that it is partially correct

This establishes the claim that the algorithm "Search by Index" is partially correct.

3

Confirm that the proposed “bound function” for the `while` loop in this algorithm really is one.

A *bound function* for a `while` loop is an integer-valued, total function of some of the inputs, variables and global data that are accessed and modified when the loop is executed — and it satisfies the following additional properties:

a) When the loop body is executed, the value of this function is decreased by at least one before the loop’s test is checked again (if it is reached again, at all)

b) If the value of this function is less than or equal to zero and the loop’s test is checked then the test fails (ending this execution of the loop)

Considering the algorithm above, we can see the following:

- `index` is an integer input and `i` is an integer variable, defined and initialized on line 5. Therefore, this is an integer-valued total function of the inputs, variables and global data that are defined when the `while` loop in this algorithm is reached
- When the body of the `while` loop is executed the value of `i` is increased by one (line 15) and the value of `index` is not changed — so the value of the proposed bound function `index-i` is decreased by one
- If the value of the proposed bound function is ≤ 0, the `while` loop’s execution would terminate

This establishes the proof for the proposed bound function.

4

Use this to complete a proof that this algorithm is correct.

If an algorithm, for a given computational problem, is **both partially correct and terminates**, whenever it is executed with its precondition initially satisfied, then this algorithm is correct.

This follows directly from the definition of *partially correct* and *termination*.

We can show that this algorithm eventually terminates:

- If the test on line 2 passes, this will lead to the `IndexOutOfBoundsException` on line 3 being thrown, and the execution of the algorithm ends
- Otherwise, it will move on to reach the `while` loop on line 13. This `while` loop has a bound function `index-i` , and the execution of this loop eventually terminates and a node is returned

Therefore, this algorithm can be considered correct as both of the conditions above are satisfied.

5

Give a bound for the number of steps executed by this algorithm — using the *uniform cost criterion* to define this — as a function of the input value `index` .

We can trace the execution for this algorithm in order to analyze its runtime, using the uniform cost criterion.

Suppose this algorithm is executed, with the problem's precondition satisfied.

- The test on line 1 costs 1 step
    - If the value of `index` is less than or equal to 0 or greater than the `length` of the list, then the execution of the algorithm halts after one more step, for a total of 2 steps
- Otherwise, the execution continues to lines 5 and 6, each of which cost 1 step
- We reach the `while` loop on line 13
    - The body of the loop (lines 14-15) gets executed *at most* `index` times, while the loop test itself (line 13) gets executed *at most* `index+1` times.
        - Each execution of the loop test requires one step and each execution of the loop body requires two steps. Thus, the total number of steps required *at most* for the `while` loop is:

$$(\texttt{index+1} \cdot 1) + (\texttt{index} \cdot 2) = 3\,\texttt{index} + 1$$

- One last step is executed after the `while` loop, which is the `return` statement on line 17

Therefore, the upper bound for the number of steps executed by this algorithm is given by:

$$(3\,\texttt{index} + 5)$$

6

Give a bound for the *worst-case* running-time of this algorithm, that is, a bound for this as a function of the length `n` of the linked list.

`index` cannot be greater than n, the length of the linked list, but it can be equal to n. Therefore, the *worst-case* running time of this algorithm is always going to be $3n + 5$, as this is the upper bound for the number of steps used in this algorithm as described above.

7

Compare this to the worst-case running to search by index in a *bounded array* or `ArrayList`. You should find that the worst-case cost for this operation in a bounded array or `ArrayList` is significantly lower.

Accessing an element in a bounded array or `ArrayList` can be done in *constant time*, given the index of this element. In other words, the worst-case running time for performing the same operation (search by index) on a bounded array or `ArrayList` is indeed much lower than the worst-case running time when performing it on a linked list.

8

If possible, describe one or more operations that you would be able to carry out in a linked list, in such a way that the *worst-case* running time of each of these operations would be significantly lower than the *worst-case* cost of the corresponding operation on a bounded array or `ArrayList`.

Placing a new node at the end of a linked list through the `add()` method would result in constant time, whereas the same operation for a bounded array or `ArrayList` can cost the entire size of the `ArrayList` in the worst-case if its size is equal to its current capacity.