EECS 2011 M: Fundamentals of Data Structures

Suprakash Datta
Office: LAS 3043

Course page: `http://www.eecs.yorku.ca/course/2011M`
Also on Moodle

Loop Invariants

Ch. 4.4, page 181

- Key idea in proving correctness of iterations
- Useful in later Algorithms and Software Engg courses
- Not dealt with in detail in our text

Note: Some slides in this presentation have been adapted from Prof Elder's slides.

Correctness Definition: Program/Method

- Input/output specifications: E.g. for sorting:
  INPUT: $A[1..n]$ - an array of integers
  OUTPUT: a permutation B of A such that
  $B[1] \leq B[2] \leq \ldots \leq B[n]$
- An input is valid if it satisfies the input specifications
- CORRECTNESS: The algorithm satisfies the output specs for EVERY valid input
  To show that the algorithm works correctly for all valid inputs of all sizes:
  - Exhaustive testing not feasible.
  - Analytical techniques are ~~useful~~ essential here.

Correctness Definition: Code Segment

- $\langle pre - condition \rangle \wedge \langle code \rangle \Rightarrow \langle post - condition \rangle$
- If the input meets the preconditions, then the output must meet the post-conditions.
- If the input does not meet the preconditions, then nothing is required.

Assertions

- An assertion is a statement about the state of the program at a specified point in its execution
- May be implemented in code, as an error-check
- Types:
  - Preconditions: Any assumptions that must be true about the code that follows
  - Postconditions: The statement of what must be true about the preceding code
  - Exit condition: The statement of what must be true to exit a loop or a method or program
  - Loop invariants: Some property that holds in each iteration of the loop, and is useful for proving correctness of the loop

Uses

- If the assertions can be checked automatically, correctness checking can be automated
- Caveat: undecidability issues
- EECS 3311 will teach you to do this in practice

Loop Invariants

- Any property that holds during each iteration of a loop
- $1 + 1 = 2, 1 \neq 0$ are valid loop invariants for any loop!
- We want to use loop invariants that help us to prove correctness of loops

Loop Invariants Example: FindMax

Input: $A[1..n]$ - an array of integers
Output: an element m of A such that $A[j] \leq m$, $1 \leq j \leq n$

FindMax(A)

$n \leftarrow length(A)$
$max \leftarrow A[1]$
for $j \leftarrow 2$ **to** n
do if $max < A[j]$
  then $max \leftarrow A[j]$
return max

Some loop invariants for the for-loop are......?

Correctness Proofs for Loops

Decompose the job into these parts

- Pre-condition for the loop
- Loop Invariant for each iteration
- Termination condition
- Termination implies post-condition

Note the similarities with induction.

Correctness of FindMax

- Pre-condition for the loop: *max* contains $A[1]$
- Loop Invariant for each iteration: At the beginning of iteration j of the for loop, *max* contains the maximum of $A[1..j-1]$
- Termination condition: $j = length(A) + 1$
- Termination implies post-condition: *max* is the correct maximum

Proof of the Loop Invariant - 1

Partial Correctness

- Initialization: max contains $A[1]$, so $LI(1)$ is true
- Maintenance: For $j > 2$, assume $LI(j-1)$; so before iteration $j-1$, $max =$ maximum of $A[1..j-2]$

  Case 1: $A[j-1] =$ maximum of $A[1..j-1]$. In lines 3,4, max is set to $A[j-1]$

  Case 2: $A[j-1]$ is not the maximum of $A[1..j-1]$, so the maximum of $A[1..j-1]$ is in $A[1..j-2]$. By our assumption, max already has this value, and max is unchanged in this iteration.

Proof of the Loop Invariant - Termination

Loop Invariant for each iteration: At the beginning of iteration j of the for loop, *max* contains the maximum of $A[1..j-1]$

- Termination: When the loop terminates, $j = length(A) + 1$
- Termination implies post-condition: *max* contains the maximum of $A[1..length(A)]$
  Therefore, it is the correct maximum

Loop Invariants - Summary

We must show three things about loop invariants:

- Initialization – it is true prior to the first iteration
- Maintenance – if it is true before an iteration, it remains true before the next iteration
- Termination – when loop terminates the invariant gives a useful property to show the correctness of the algorithm

$\boxed{\text{Partial Correctness} \wedge \text{Termination} \Rightarrow \text{Correctness}}$

Binary Search

- Preconditions: Given a key(25), a sorted list of keys

3	5	6	13	18	21	21	25	36	43	49	51	53	60	72	74	83	88	91	95

- PostCondition: Find key in list (if there)

3	5	6	13	18	21	21	25	36	43	49	51	53	60	72	74	83	88	91	95

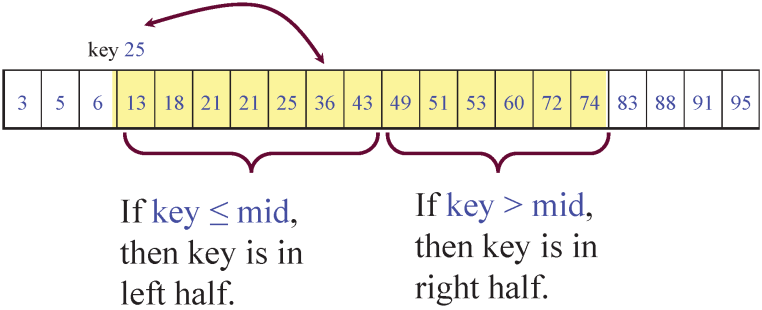
- Define a loop invariant:
  - Maintain a sublist
  - If the key is contained in the original list, then the key is contained in the sublist

Binary Search: Loop Invariant

- Cut sublist in half
- Determine which half the key would be in
- Keep that half
- Caveat: Invariant must not assume that the element is present in the list.
  So it should say something like
  "If the key is contained in the original list, then the key is contained in the sublist."

Binary Search: Algorithm Design

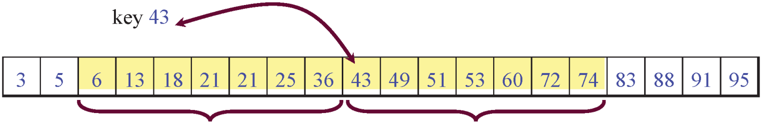
- It is faster not to check if the middle element $=$ *key*



- The size of the list gets smaller
- If the sublist has even length, which element is mid? Does not matter – choose right.

Binary Search: Mistakes

- If $key \leq mid$, then key is in left half: $[i, mid - 1]$
  If $key > mid$, then key is in right half: $[mid, j]$

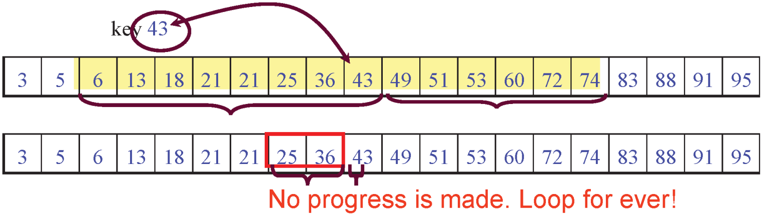


If the middle element is the key, it can be skipped over!

- Possible fix?
  If $key < mid$, then key is in left half: $[i, mid - 1]$
  If $key \geq mid$, then key is in right half: $[mid, j]$

Binary Search: Another Mistake

- Possible fix: making the left half slightly bigger.
- If $key \leq mid$, then key is in left half: $[i, mid]$
  If $key > mid$, then key is in right half: $[mid + 1, j]$



Binary Search: pseudo-code

algorithm $BinarySearch\,(\langle L(1..n), key\rangle)$

$\langle$***pre−cond***$\rangle$: $\langle L(1..n), key\rangle$ is a sorted list and key is an element.

$\langle$***post−cond***$\rangle$: If the key is in the list, then the output consists of an index i such that $L(i) = key$.

```
begin
    i = 1, j = n
    loop
        ⟨loop−invariant⟩: If the key is contained in L(1..n), then
            the key is contained in the sublist L(i..j).

        exit when j ≤ i
        mid = ⌊(i+j)/2⌋
        if(key ≤ L(mid)) then
            j = mid          % Sublist changed from L(i,j) to L(i..mid)
        else
            i = mid + 1      % Sublist changed from L(i,j) to L(mid+1,j)
        end if
    end loop
    if(key = L(i)) then
        return( i )
    else
        return( "key is not in list" )
    end if
end algorithm
```

Another Example

algorithm $GCD(a,b)$

$\langle \textbf{\textit{pre}} - \textbf{\textit{cond}} \rangle$**:** a and b are integers.

$\langle \textbf{\textit{post}} - \textbf{\textit{cond}} \rangle$**:** Returns $GCD(a, b)$.

begin
  int x,y
  $x = a$
  $y = b$
  loop
    $\langle \textbf{\textit{loop}} - \textbf{\textit{invariant}} \rangle$**:** GCD($x$,$y$) = GCD(a,b).

    if($y = 0$) exit
    $x_{new} = y \quad y_{new} = x \ mod \ y$
    $x = x_{new}$
    $y = y_{new}$
  end loop
  return(x)
end algorithm