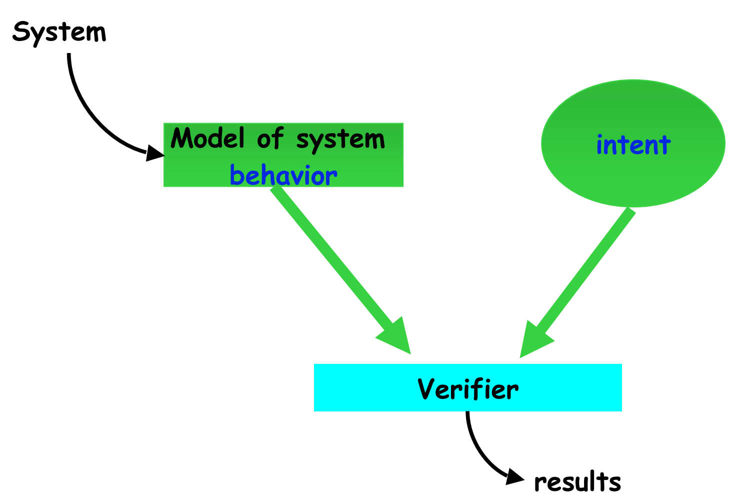
Formal Verification

Basic Verification Strategy

compare behavior to intent



Intent

- Usually, originates with requirements, refined through design and implementation
- formalized by specifications
  - Often expressed as formulas in mathematical logic
- different types of intent
  - E.g.,performance, functional behavior
  - each captured with different types of formalisms
  - specification of behavior/functionality
    - what functions does the software compute?
    - Often expressed using predicate logic

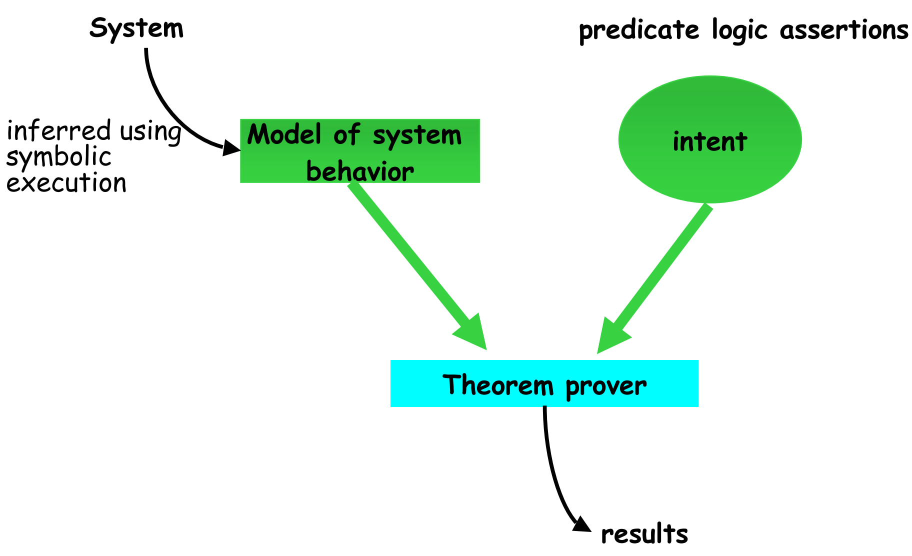
Compare behavior to intent

- can be done informally- by human eye
  - Cleanroom
  - Inspections
- can be done selectively
  - Checking assertions during execution
- can be done formally
  - With theorem proving
    - Usually with automated support
    - Called Proof of Correctness or Formal Verification
      - Proof of "correctness" is dangerously misleading
  - With static analysis for restricted classes of properties

Theorem Proving based Verification

- Behavior inferred from semantically rich program model
  - generally requires most of the semantics of the programming language
  - employs symbolic execution
- Intent captured by predicate calculus specifications (or another mathematically formal notation)

Theorem-Proving based Verification Strategy



Floyd Method of Inductive Assertions

- Show that given input assertions, after executing the program, program satisfies output assertions
  - show that each program fragment behaves as intended
  - use induction to prove that all fragments, including loops, behave as intended
- show that the program must terminate

Mathematical Induction

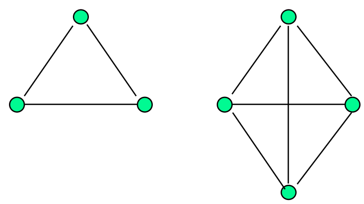
- goal: prove that a given property holds for all elements of a set
- approach:
  - show property holds for "first" element
  - show that if property holds for element i, then it must also hold for element i + 1
- often used when direct analytic techniques are too hard or complex

Example: How many edges in C_n

Theorem:

let $C_n = (V_n, E_n)$ be a complete, unordered graph on n nodes,

then $|E_n| = n * (n-1)/2$



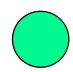
Example: How many edges in C_n

- to show that this property holds for the entire set of complete graphs, $\{C_i\}$, by induction:
  1. show the property is true for C_1
  2. show if the property is true for C_n, then the property is true for C_{n+1}

Example: How many edges in C_n

show the property is true for C1:

graph has one node, 0 edges



$$|E_1| = n(n-1)/2 = 1(0)/2 = 0$$

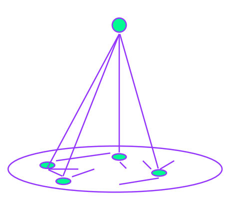
Example: How many edges in C_n

assume true for C_n: $|E_n| = n(n-1)/2$

graph C_{n+1} has one more node, but n more edges (one from the new node to each of the n old nodes)

Thus, want to show $|E_{n+1}| = |E_n| + n = (n+1)(n)/2$

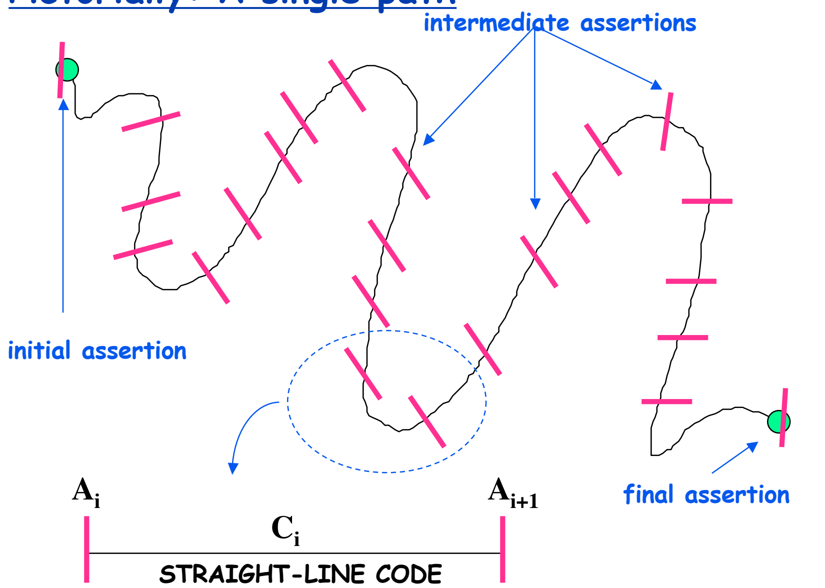
Proof: $|E_{n+1}| = |E_n| + n = n(n-1)/2 + n$	by substitution
$= n(n-1)/2 + 2n/2$	by rewriting
$= (n(n-1)+2n)/2$	by simplification
$= (n(n-1+2))/2$	by simplification
$= n(n+1)/2$	by simplification
$= (n+1)(n)/2$	by rewriting



Floyd's Method of inductive verification (informal description)

- Place assertions at the start, final, and intermediate points in the code.
- Any path is composed of sequences of program fragments that start with an assertion, are followed by some assertion free code, and end with an assertion
  - A_s, C_1, A_2, C_2, A_3,…A_{n-1}, C_{n-1}, A_f
- Show that for every executable path, if A_s is assumed true and the code is executed, then A_f is true

Pictorially: A single path

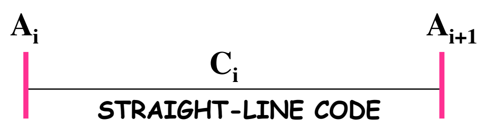


Must be sure:

assuming A_i,

then executing Code C_i,

necessarily $\Rightarrow$ $A_i + 1$



by forward substitution

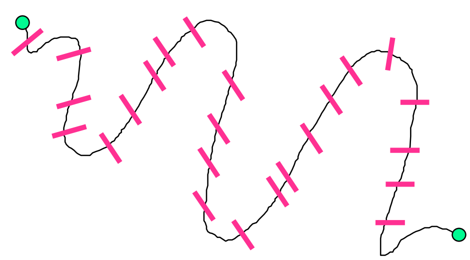
$\Rightarrow$ symbolic execution

Why does this work?

suppose P is an arbitrary path through the program

can denote it by

$P = A_0\ C_1\ A_1\ C_2\ A_2 \ldots C_n\ A_n$



where

- A_0 - Initial assertion
- A_n - Final assertion
- A_i - Intermediate assertions
- C_i - Loop free, uninterrupted, straight-line code

If it has been shown that

$\forall$ i, $1 \leq i < n$: $A_i C_i \Rightarrow A_{i+1}$

Then, by transitivity

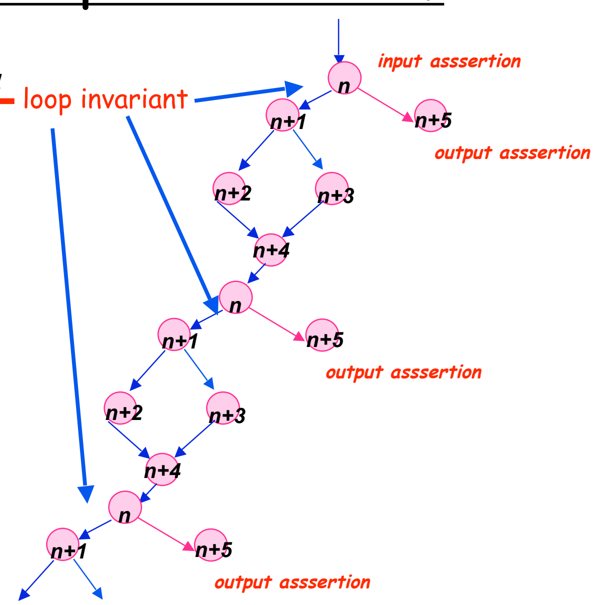
$A_0 \ldots\ldots \Rightarrow A_n$

Obvious problems

- How do we do this for a path?
- How do we do this for all paths?
  - Infinite number of paths
    - Must find a way to deal with loops

How to handle loops -- unroll them

```
            input asssertion
n           do_while predicate1
n+1             if predicate2
n+2                 then code ;
n+3                 else code ;
n+4                     end;
n+5         output assertion ;
```



Better -- find loop invariant (A_I)

subpaths to consider:

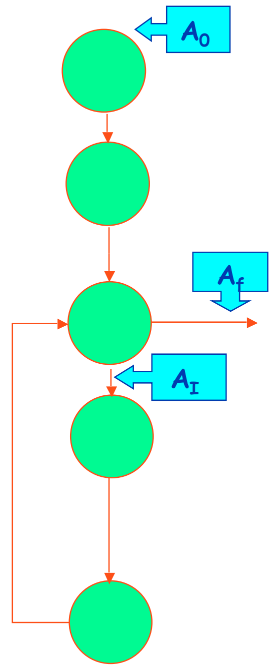
C_1: Initial assertion A_0 to final assertion A_f

C_2: Initial assertion A_0 to A_I

C_3: A_I to A_I

C_4: A_I to final assertion A_f

Similar to an inductive proof



Consider all paths through a loop

subpaths to consider:

C_1: A_0 to A_f

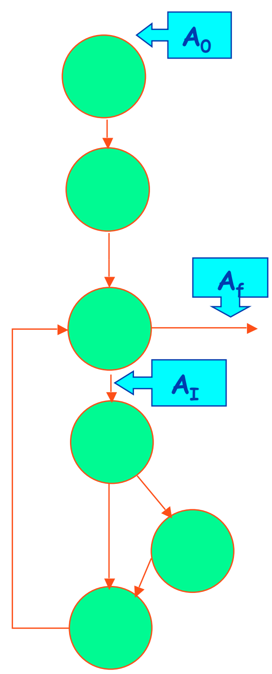
C_2: A_0 to A_I

C_3: A_I, false branch, A_I

C_4: A_I, true branch, A_I

C_5: A_I, false branch, A_f

C_6: A_I, true branch, A_f



Assertions

- specification that is intended to be true at a given site in the program
- Use three types of assertions:
  - initial : sited before the initial statement
  - final : sited after the final statement
  - intermediate: sited at various internal program locations subject to the rule:
    - a "loop invariant" is true on every iteration thru the loop

Floyd's Inductive Verification Method (more carefully stated)

- specify initial and final assertions to capture intent
- place intermediate assertions so as to "cut" every program loop
- For each pair of assertions where there is at least one executable (assertion-free) path from the first to the second,
  - assume that the first assertion is true
  - show that for all (assertion-free, executable) paths from the first assertion to the second, that the second assertion is true
  - This establishes "partial correctness"
- Show that the program terminates
  - This establishes "total correctness"

Example

- **Assume we have a method, called FindValue, that takes as input three parameters: a table that is an array of values where the index starts at zero, n is the current number of values in table (with entries from 0 to n-1), and a key that is also of type value. FindValue returns the smallest index of the element in table that is equal to the value of key. If no element of table is equal to key, then a new last element with that value is added to the table and that index is returned.**

```
// preconditions
requires n >= 0;
requires key != null;
requires table != null;
requires n<=table.size();
```

```
// postconditions
ensures (table[\result] == key)
ensures \forall(int i=0;i < \result; i++;)
        (table[i] != key)
ensures \result>=0 && \result<=n
ensures \result==n => table.size()>=n+1
ensures \result<n => table.size() >=n
```

Example: FindValue version 1

```
Boolean FindValue (int table[ ], int n, int key) {
  boolean found;
  found=false;
  current = 0;
  while (not found && current <  n) {
    if (table[current] == key)
          found = true;
    else
          current = current + 1;
  }
  if (not found) {
    table[current] = key;
  }
  return (current);
}
```

```
// preconditions
requires n >= 0;
requires key != null;
requires table != null;
requires n<=table.size();
```

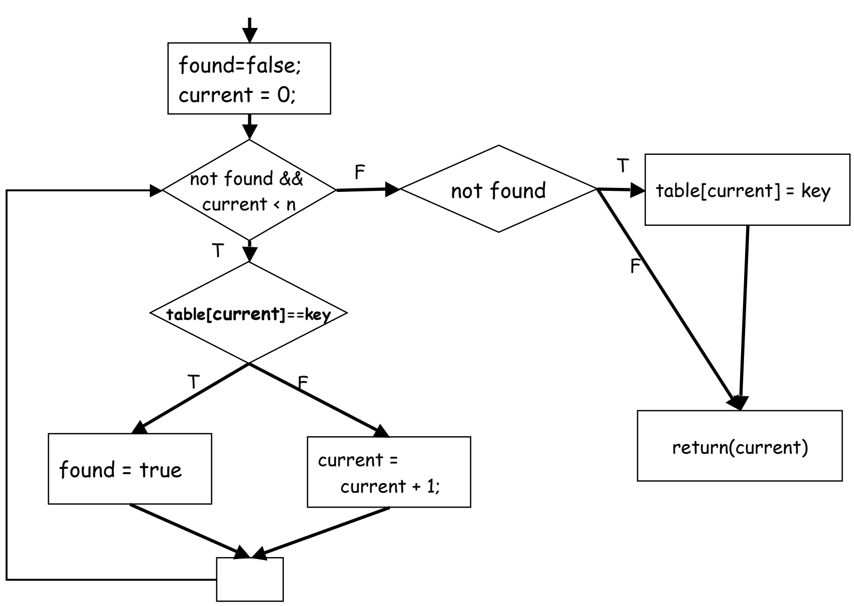
```
// postconditions
ensures (table[\result] == key)
ensures \forall(int i=0;i < \result; i++;)
        (table[i] != key)
ensures \result>=0 && \result<=n
ensures \result==n => table.size()>=n+1
ensures \result<n => table.size() >=n
```

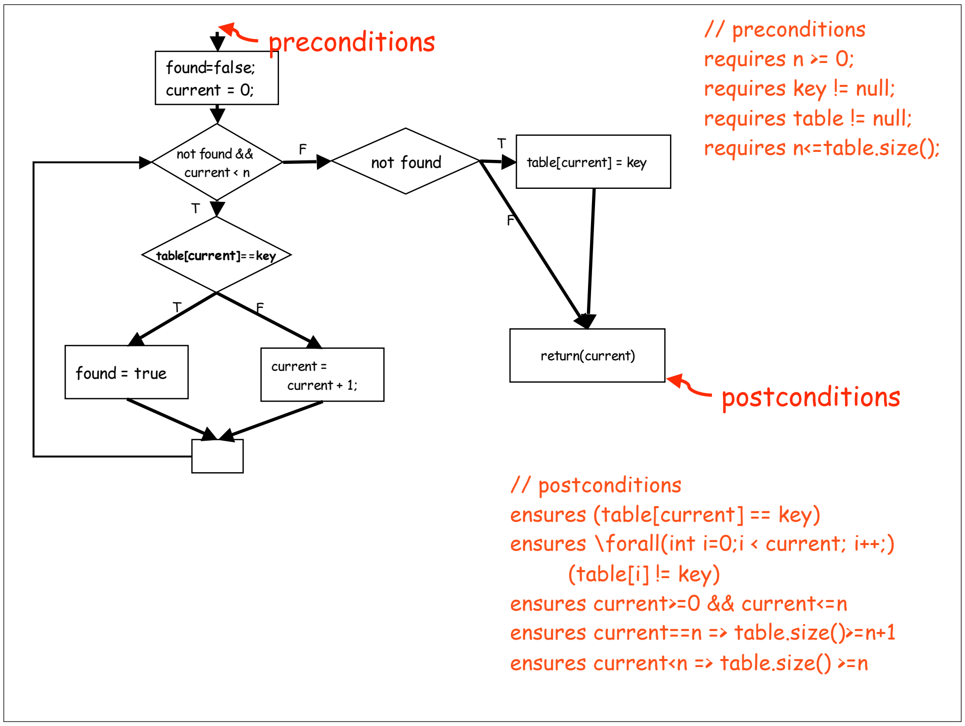
Example: FindValue version 2

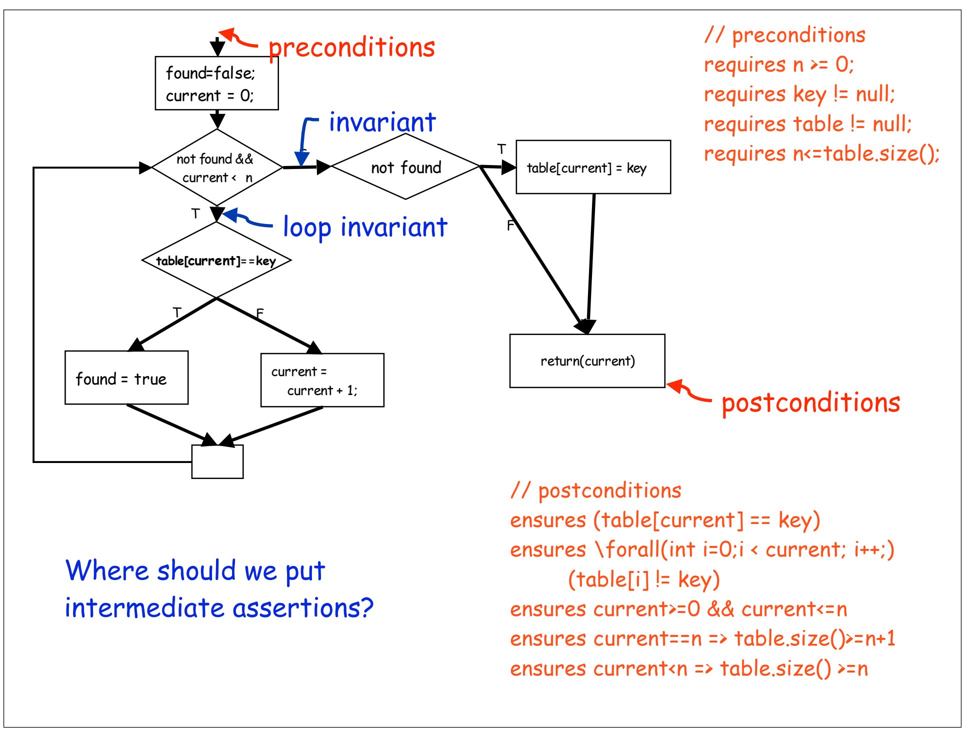
```
Boolean FindValue (int table[ ], int n, int key) {
current = 0;
while (table[current] != key && current< n) {
    current = current + 1;
}
if (current = n) {
   table[current] = key;
   }
return (current);
}
```

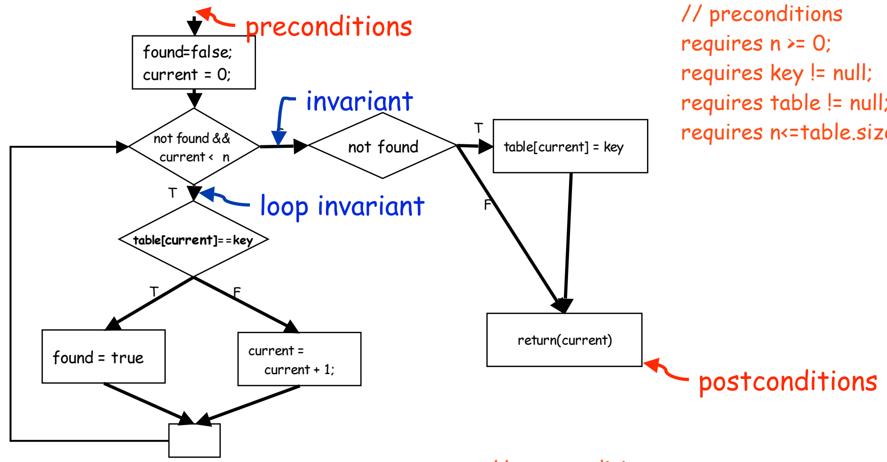
```
// preconditions
requires n >= 0;
requires key != null;
requires table != null;
requires n<=table.size();
```

```
// postconditions
ensures (table[\result] == key)
ensures \forall(int i=0;i < \result; i++;)
        (table[i] != key)
ensures \result>=0 && \result<=n
ensures \result==n => table.size()>=n+1
ensures \result<n => table.size() >=n
```




preconditions
found=false;
current = 0;
not found &&
current < n
F
T
not found
T
table[current] = key
F
table[current]==key
T
F
found = true
current =
current + 1;
return(current)
postconditions
// preconditions
requires n >= 0;
requires key != null;
requires table != null;
requires n<=table.size();
// postconditions
ensures (table[current] == key)
ensures \forall(int i=0;i < current; i++;)
(table[i] != key)
ensures current>=0 && current<=n
ensures current==n => table.size()>=n+1
ensures current<n => table.size() >=n

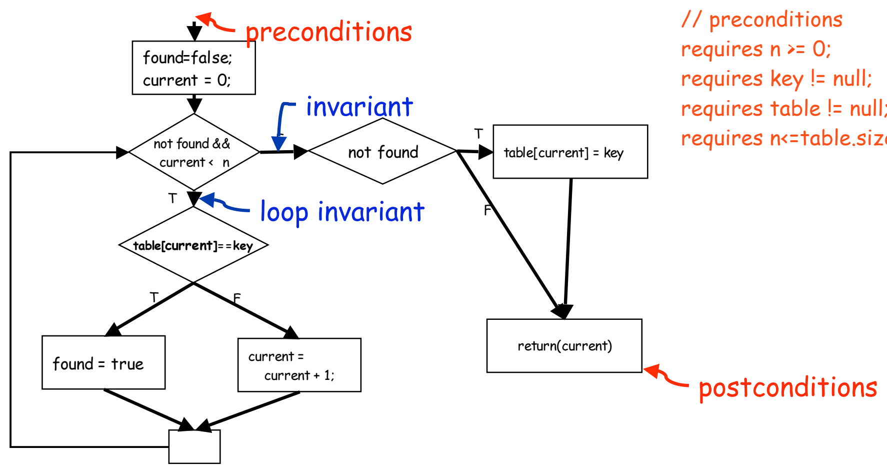

preconditions
found=false;
current = 0;
invariant
not found &&
current < n
T
loop invariant
not found
T
table[current] = key
F
table[current]==key
T
F
found = true
current =
current + 1;
return(current)
postconditions
// preconditions
requires n >= 0;
requires key != null;
requires table != null;
requires n<=table.size();
// postconditions
ensures (table[current] == key)
ensures \forall(int i=0;i < current; i++;)
(table[i] != key)
ensures current>=0 && current<=n
ensures current==n => table.size()>=n+1
ensures current<n => table.size() >=n
Where should we put
intermediate assertions?



// preconditions
requires n >= 0;
requires key != null;
requires table != null;
requires n<=table.size();

What do we need to prove?:
precondition {code1} invariant
precondition {code2} loop invariant
loop invariant {code3} loop invariant: 2 cases?
loop invariant {code4} invariant: 2 cases?
invariant {code 5} postconditions:2 cases

// postconditions
ensures (table[current] == key)
ensures \forall(int i=0;i < current; i++;)
(table[i] != key)
ensures current>=0 && current<=n
ensures current==n => table.size()>=n+1
ensures current<n => table.size() >=n



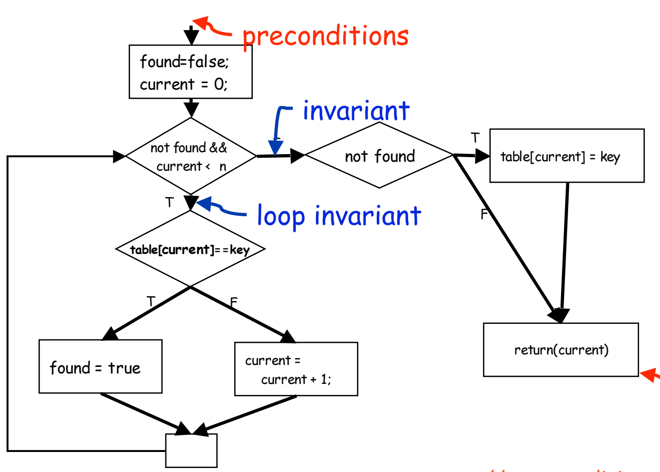
// preconditions
requires n >= 0;
requires key != null;
requires table != null;
requires n<=table.size();

What do we need to prove?:
precondition {code1} invariant
precondition {code2} loop invariant
loop invariant {code3} loop invariant
loop invariant {code4} invariant:
invariant {code 5} postconditions:2 cases

verification conditions

What needs to be done

- **Must define all the intermediate assertions**
- **Must create all the verification conditions**
- **Must prove each verification condition**
- **Must prove termination**

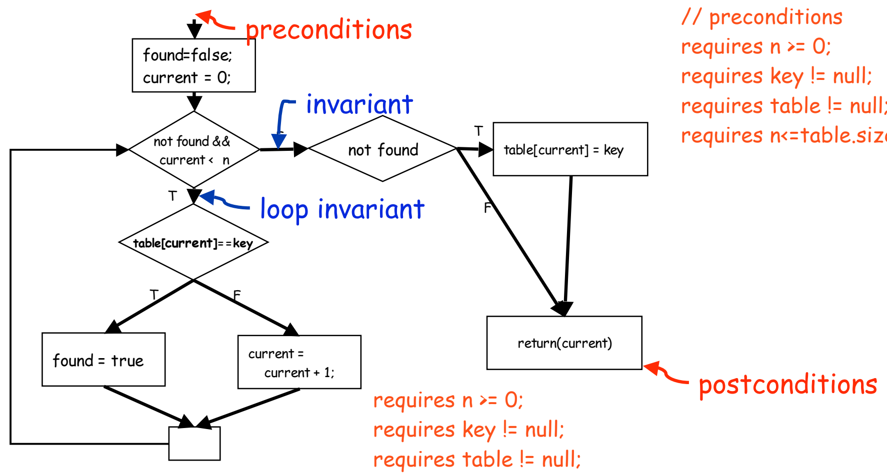


// preconditions
requires n >= 0;
requires key != null;
requires table != null;
requires n<=table.size()+1;

What should the loop invariant be?

ensures (found ==false)
ensures current>=0 && current<n
ensures \forall(int i=0;i < current; i++;)
    (table[i] != key)

// postconditions
ensures (table[current] == key)
ensures \forall(int i=0;i < current; i++;)
    (table[i] != key)
ensures current>=0 && current<=n
ensures current==n => table.size()>=n+1
ensures current<n => table.size() >=n



// preconditions
requires n >= 0;
requires key != null;
requires table != null;
requires n<=table.size();

What do we need to prove?:

precondition {code2} loop invariant

requires n >= 0;
requires key != null;
requires table != null;
requires n<=table.size();
{found= false
current =o
not found &&
Current <n}
ensures (found ==false)
ensures current>=0 && current<n
ensures \forall(int i=0;i < current; i++;) (table[i] != key)

Proving one verification condition

precondition {code2} loop invariant

requires n >= 0;
requires key != null;
requires table != null;
requires n<=table.size();
{found= false
current =o
not found &&
current < n;}
ensures (found ==false)
ensures current>=0 && current< n
ensures \forall(int i=0;i < current; i++;)
(table[i] != key)

Proof:precondition {code2} loop invariant

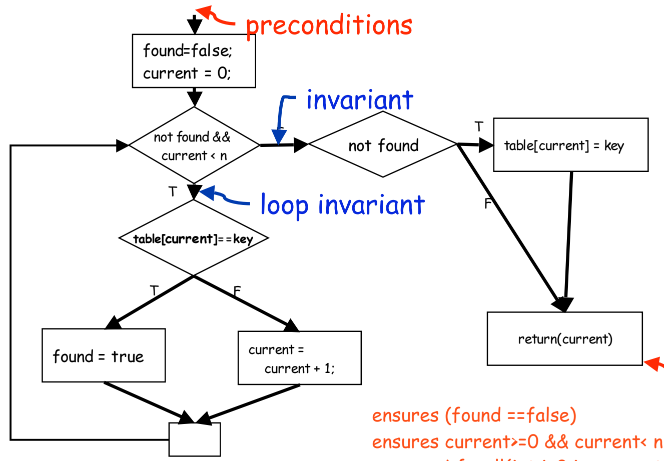
Executing

{found= false
current =o
not found &&
current < n}

a) By execution found ==false

b) By execution, current =0 and
given n>=0 => n > current=0
therefore
current>=0 && current< n

c) int i=0;i < 0 is empty, so stmt
(table[i] != key) is true



// preconditions
requires n >= 0;
requires key != null;
requires table != null;
requires n<=table.size();

postconditions

ensures (found ==false)
ensures current>=0 && current< n
ensures \forall(int i=0;i < current; i++;) (table[i] != key)

precondition {code1} invariant
precondition {code2} loop invariant
loop invariant {code3} loop invariant
loop invariant {code4} invariant
invariant {code 5} postconditions:2 cases

{not found &&
current < n
If table[current]==key then
found =true
else current =current +1
not found && current < n
}
loop invariant again

Need to represent old versus new values

Proving another verification condition

loop invariant {code3} loop invariant
ensures (found' ==false)
ensures current'>=0 && current'< n
ensures \forall(int i=0;i < current'; i++;)
(table[i] != key)

{not found' && current' < n
If table[current'] == key then
found =true
else current =current' +1
not found && current < n
}
ensures (found ==false)
ensures current>=0 && current< n
ensures \forall(int i=0;i < current; i++;)
(table[i] != key)

Proof: loop invariant {code3} loop invariant

a) By execution, not found = true =>
found = false

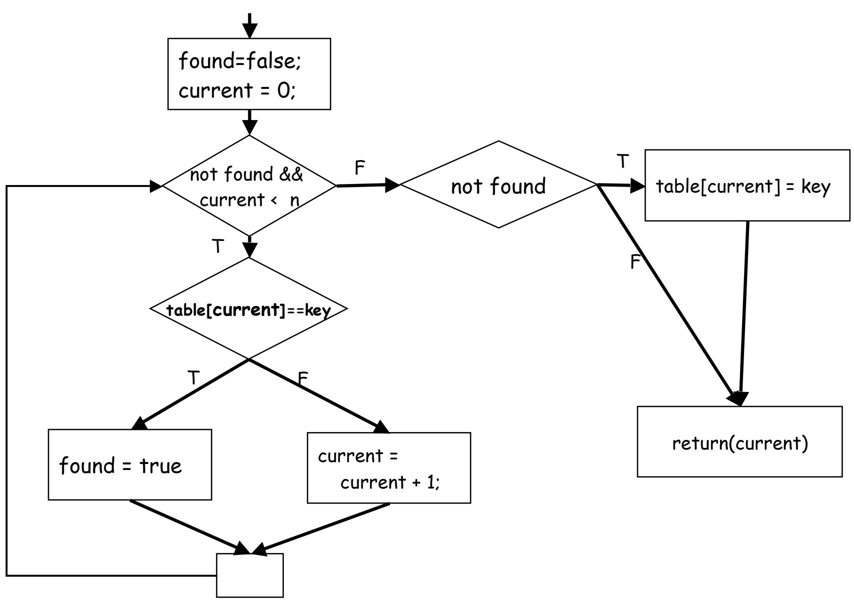
b) By execution, (current < n) = true =>
current <=n
Given current'>= 0 and by execution that
current =current' +1 => current>=0

therefore current>=0 && current< n

c) Given forall(int i=0;i < current'; i++;)
(table[i] != key)
Given found = false by (a) above then
in execution
table[current'] != key
and current =current' +1
=> forall(int i=0;i < current'+1; i++;)
(table[i] != key)
=> forall(int i=0;i < current; i++;)
(table[i] != key)

What remains to be done?

- **Must prove all the verification conditions**
  - **precondition {code1} invariant**
  - **precondition {code2} loop invariant**
  - **loop invariant {code3} loop invariant**
  - **loop invariant {code4} invariant**
  - **invariant {code 5} postconditions:2 cases**
- **Must prove termination**



Floyd-Hoare axiomatic proof method

assertions are preconditions and postconditions on some statement or sequence of statements

P{S}Q

if P is true before S is executed and S is executed then Q is true

P is the precondition;
Q is the postcondition

Also written {P} S {Q}

Floyd-Hoare axiomatic proof method

- as in Floyd's inductive assertion method, we construct a sequence of assertions, each of which can be inferred from previously proved assertions and the rules and axioms about the statements and operations of the program
- to prove P{S}Q, we need some axioms and rules about the programming language

Hoare axioms and proof rules

take a simple programming language that deals only with integers and has the following types of constructs:

- assignment statement
  x:= f
- composition of a sequence of statements
  S1, S2
- conditional (alternative statements)
  if B then S1 else S2
- iteration
  while B do S

Axioms and proof rules

- axiom of assignment

  P {x:=f} Q,

  where Q is obtained from P by substituting f for all occurrences of x in P (symbolic execution)
- rule of composition

  P {S1, S2 } Q => ∃ P1 , P{S1}P1 ∧ P1{S2}Q

  Using Hoare's notation, this is written as

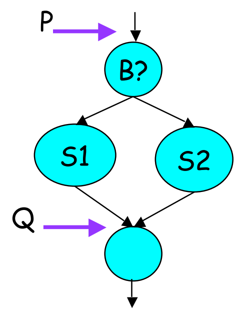
$$\frac{P\{S1\}P1,\ P1\{S2\}Q}{P\ \{S1,S2\}\ Q}$$

Proof Rules (continued)

- rule for the alternative statement

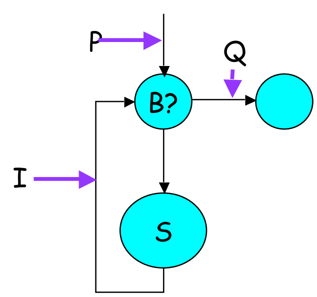
  P{if B then S1 else S2 }Q ⇒
  P{B ∧ S1}Q ∧ P{~B ∧ S2}Q

- Hoare's notation



$$\frac{P\{B \wedge S1\}Q,\ P\{\neg B \wedge S2\}Q}{P\{\text{if } B \text{ then } S1 \text{ else } S2\}Q}$$

Proof Rules (continued)



rule of iteration

P {while B do S }Q ⇒
P{¬B}Q ∧ ∃ I ∍ P {B ∧ S} I
∧ I{B ∧ S } I ∧ I{¬B }Q

$$\frac{P\{\neg B\}Q,\ P\{B \wedge S\}\ I,\ I\{B \wedge S\}\ I,\ I\{\neg B\}Q}{P\ \{\text{while } B \text{ do } S\}Q}$$

weakest precondition

- in Hoare technique P{S}Q

```
S1:
read x,y;
z:= y
while x >0 do
   z:= z+1;
   x:= x-1;
endwhile;
```

```
S2:
read x,y;
z:= x+y;
```

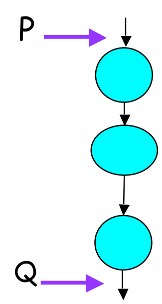
suppose $P = \{x \geq 0\}$

$Q = \{z = x+y\}$

- then we can prove P{S1}Q and P{S2}Q, but we can also prove true{S2}Q
- S2 is provable for any x, y, but S1 is provable only for $x \geq 0$

Dijkstra's Axiomatic Semantics

- In general, there are many correct pre- and post-conditions for a given program
- Seek the strongest post condition and the weakest precondition
  - A ⇒ B; A is stronger than B and B is weaker than A



Rules of consequence

- If $P \Rightarrow P'$ and $Q' \Rightarrow Q$ and $P'\{S\}Q'$ then $P\{S\}Q$

$$\frac{P\{S\}Q',\ Q' \Rightarrow Q}{P\{S\}Q}$$

$$\frac{P \Rightarrow P',\ P'\{S\}Q}{P\{S\}Q}$$

$$\frac{P \Rightarrow P',\ P'\{S\}Q',\ Q' \Rightarrow Q}{P\{S\}Q}$$

Formal Verification Process

- determine input, output and loop invariant assertions
- identify all paths between two assertions (with no intervening assertions) and form the corresponding verification condition or lemma
- prove each verification condition (partial correctness)
- prove that the program terminates