CSE 12:

Basic data structures and object-oriented design

Jacob Whitehill
jake@mplab.ucsd.edu

Lecture Ate
11 Aug 2011

More on performance analysis.

Asymptotic performance analysis

- Asymptotic performance analysis is a coarse but useful means of describing and comparing the performance of algorithms as a function of the input size *n* when *n* gets large.
- Asymptotic analysis applies to both **time cost** and **space cost**.
- Asymptotic analysis hides details of timing (that we don't care about) due to:
  - Speed of computer.
  - Slight differences in implementation.
  - Programming language.

O, Ω, and θ

- In order to justify describing the time cost $T(n)$ $=3n+4$ as just "linear" (n), we first need some mathematical machinery:
  - We define a *lower bound* on T with Ω.
  - We define an *upper bound* on T with O.
  - We define a *tight bound* (bounded above and below) on T with θ.
    - θ is important because it is more specific than O.
      (For example, technically, $3n+4=O(2^n)$.)

Abuse of notation

- When we say that $3n+5$ is “linear in n”, what we really mean (mathematically) is that $3n+5$ is $\theta(n)$.
- *Note*: In computer science, we often say O where we really mean θ. This is a slight *abuse of notation*.
  - We will use O in this course to mean θ.

Asymptotic performance analysis

- Asymptotic analysis assigns algorithms to different "complexity classes":
  - $O(1)$ - constant - performance of algorithm does not depend on input size.
  - $O(n)$ - linear - doubling n will double the time cost.
  - $O(\log n)$ - logarithmic
  - $O(n^2)$ - quadratic
  - $O(2^n)$ - exponential
- Algorithms that differ in complexity class can have *vastly* different run-time performance (for large n).

Analysis of data structures

- Let's put these ideas into practice and analyze the performance of algorithms related to **`ArrayList`**:
  - **`add(o)`**, **`get(index)`**, **`find(o)`**, and **`remove(index)`**.
- As a first step, we must decide what the "input size" means.
  - What is the "input" to these algorithms?

Analysis of data structures

- Each of the methods (algorithms) above operates on the `_underlyingStorage` *and* either `o` or `index`.
  - `o` and `index` are always length 1 -- *their size cannot grow*.
  - However, the number of data in `_underlyingStorage` (stored in `_numElements`) will grow as the user adds elements to the `ArrayList`.
- Hence, we measure asymptotic time cost as a function of *n*, the number of elements stored (`_numElements`).

Adding to back of list

- What is the time complexity of this method?

```
class ArrayList<T> {
  ...
  void addToBack (T o) {
    // Assume _underlyingStorage is big enough
    _underlyingStorage[_numElements] = o;
    _numElements++;
  }
}
```

Adding to back of list

- What is the time complexity of this method?

Note that, for this method, the worst case, average case, and best case are all the same.

```
class ArrayList<T> {
  ...
  void addToBack (T o) {
    // Assume _underlyingStorage is big enough
    _underlyingStorage[_numElements] = o;
    _numElements++;
  }
}
```

O(1) -- no matter how many elements the list already contains, the cost is just 2 “abstract operations”.

Retrieving an element

- What is the time complexity of this method?

```
class ArrayList<T> {
  ...
  T get (int index) {
    return _underlyingStorage[index];
  }
}
```

Retrieving an element

- What is the time complexity of this method?

```
class ArrayList<T> {
  ...
  T get (int index) {
    return _underlyingStorage[index];
  }
}
```

O(1).

Adding to front of list

- What is the time complexity of this method?

```
class ArrayList<T> {
   ...
   void addToFront (T o) {
      ...
   }
}
```

Adding to front of list

- What is the time complexity of this method?

```
class ArrayList<T> {
  ...
  void addToFront (T o) {
    // Assume _underlyingStorage is big enough
    for (int i = 0; i < _numElements; i++) {
      _underlyingStorage[i+1] = _underlyingStorage[i];
    }
    _underlyingStorage[i] = o;
    _numElements++;
  }
}
```

We have to move everything over by 1.

$O(n)$.

Finding an element

- What is the time complexity of this method in the *best case*? *Worst case*?

```
class ArrayList<T> {
  ...
  // Returns lowest index of o in the ArrayList, or
  // -1 if o is not found.
  int find (T o) {
    for (int i = 0; i < _numElements; i++) {
      if (_underlyingStorage[i].equals(o)) { // not null
        return i;
      }
    }
    return -1;
  }
}
```

Finding an element

- What is the time complexity of this method in the *best case*? *Worst case*?

```
class ArrayList<T> {
  ...
  // Returns lowest index of o in the ArrayList, or
  // -1 if o is not found.
  int find (T o) {
    for (int i = 0; i < _numElements; i++) {
      if (_underlyingStorage[i].equals(o)) { // not null
        return i;
      }
    }
    return -1;
  }
}
```

$O(1)$ in best case; $O(n)$ in worst case.

Adding *n* elements

- Now, let's consider the time complexity of doing *many adds in sequence*, starting from an empty list:

```
void addManyToFront (T[] many) {
  for (int i = 0; i < many.length; i++) {
    addToFront(many[i]);
  }
}
```

- What is the time complexity of **`addManyToFront`** on an array of size *n*?

Adding *n* elements

- To calculate the total time cost, we have to *sum* the time costs of the individual calls to `addToFront`.
  - Each call to `addToFront(o)` takes about time *i*, where *i* is the *current* size of the list. (We have to "move over" *i* elements by one step to the right.)

```
void addManyToFront (T[] many) {
  for (int i = 0; i < many.length; i++) {
    addToFront(many[i]);
  }
}
```

- Let $T(i)$ the cost of `addToFront` at iteration *i*: $T(0)=1$, $T(1)=2$, ..., $T(n-1)=n$.

Adding *n* elements

- Now we just have to add together all the *T*(*i*):

$$\sum_{i=0}^{n-1} T(i) = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$$

- Note that we would get the same asymptotic bound even if we calculated the cost *T*(*i*) slightly differently, e.g., *T*(*i*)=3*i*+2:

$$\begin{aligned}
\sum_{i=0}^{n-1} T(i) &= \sum_{i=0}^{n-1} (3i+2) \\
&= \sum_{i=0}^{n-1} 3i + \sum_{i=0}^{n-1} 2 \\
&= 3\sum_{i=0}^{n-1} i + 2n \\
&= 3\left(\frac{n(n-1)}{2}\right) + 2n \\
&= O(n^2)
\end{aligned}$$

Finding an element

- What is the time complexity of this method in the *average case*?

```
class ArrayList<T> {
  ...
  // Returns lowest index of o in the ArrayList, or
  // -1 if o is not found.
  int find (T o) {
    for (int i = 0; i < _numElements; i++) {
      if (_underlyingStorage[i].equals(o)) { // not null
        return i;
      }
    }
    return -1;
  }
}
```

Finding an element: average case

- Finding an exact formula for the *average case* performance can be tricky (if not impossible).
- In order to compute the average, or *expected*, time cost, we must know:
  - The *time cost* $T(X_n)$ for a particular *input* X of size n.
  - The *probability* $P(X_n)$ of that input X.
  - The *expected time cost*, over all inputs X of size n, is then:

$$\text{AvgCaseTimeCost}_n = E[T(X_n)] = \sum_{X_n} P(X_n)T(X_n)$$

Finding an element: average case

- Finding an exact formula for the *average case* performance can be tricky (if not impossible).
- In order to compute the average, or *expected*, time cost, we must know:

  In this case, `x` consists of both the element `o` and the contents of `_underlyingStorage`.

  - The *time cost* $T(X_n)$ for a particular *input* X of size n.
  - The *probability* $P(X_n)$ of that input X.
  - The *expected time cost*, over all inputs X of size n, is then:

$$\text{AvgCaseTimeCost}_n = E[T(X_n)] = \sum_{X_n} P(X_n)T(X_n)$$

"*E*" for "Expectation"

Sum the time costs for all possible inputs, and weight each cost by how likely it is to occur.

Finding an element: average case

- In the `find(o)` method listed above, it is possible that the user gives us an `o` that is not contained in the list.
  - This will result in $O(n)$ time cost.
  - How "likely" is this event?
    - *We have no way of knowing* -- we could make an arbitrary assumption, but the result would be meaningless.
  - Let's *remove this case from consideration* and assume that `o` is always present in the list.
    - What is the average-case time cost *then*?

Finding an element: average case

- Even when we assume `o` is present in the list somewhere, we have no idea whether the `o` the user gives us will "tend to be at the front" or "tend to be at the back" of the list.
- However, here we can make a plausible assumption:
  - For an **`ArrayList`** of n elements, the probability that o is contained at index i is $1/n$.
    - In other words, `o` is equally likely to be in any of the "slots" of the array.

Finding an element: average case

- Given this assumption, we can finally make headway.
- Let's define *T*(*i*) to be the cost of the `find(o)` method as a function of *i*, the location in `_underlyingStorage` where `o` is located. What is *T*(*i*)?

```
class ArrayList<T> {
  ...
  // Returns lowest index of o in the ArrayList, or
  // -1 if o is not found.
  int find (T o) {
    for (int i = 0; i < _numElements; i++) {
      if (_underlyingStorage[i].equals(o)) { // not null
        return i;
      }
    }
    return -1;
  }
}
```

Finding an element: average case

- Given this assumption, we can finally make headway.
- Let's define $T(i)$ to be the cost of the **`find(o)`** method as a function of i, the location in **`_underlyingStorage`** where **`o`** is located. What is $T(i)$?

```
class ArrayList<T> {
  ...
  // Returns lowest index of o in the ArrayList, or
  // -1 if o is not found.
  int find (T o) {
    for (int i = 0; i < _numElements; i++) {
      if (_underlyingStorage[i].equals(o)) { // not null
        return i;
      }
    }
    return -1;
  }
}
```

$T(i)=i$

Finding an element: average case

- Now, we can re-write the expected time cost in terms of an arbitrary input X, as the expected time cost in terms of *where in the array the element* o *will be found.*

$$
\begin{aligned}
\text{AvgCaseTimeCost}_n &= \sum_i P(i)T(i) && \text{Redefine } P(X_n) \text{ and } T(X_n) \text{ in terms of } P(i) \text{ and } T(i). \\
&= \sum_i \frac{1}{n} i && \text{Substitute terms.} \\
&= \frac{1}{n} \sum_i i && \text{Move } 1/n \text{ out of the summation.} \\
&= \frac{1}{n} \frac{n(n+1)}{2} && \text{Formula for arithmetic series.} \\
&= \frac{n+1}{2} && \text{The } n\text{'s cancel.} \\
&= O(n) && \text{Find asymptotic bound.}
\end{aligned}
$$

Questions to ponder

- What is the time cost of adding to the back of a *singly*-linked list, as a function of the number of elements already in the list?
  - With just a `_head` pointer?
  - With both `_head` and `_tail`?
  - What if `_head` and `_tail` point to dummy nodes?

More on performance measurement.

Empirical performance measurement

- As an alternative to describing an algorithm's performance with a "number of abstract operations", we can also measure its time empirically using a clock.
- As illustrated last lecture, counting "abstract operations" can anyway hide real performance differences, e.g., between using `int[]` and `Integer[]`.

Empirical performance measurement

- There are also many cases where you don't know how an algorithm works internally.
  - Many programs and libraries are not open source!
    - You have to analyze an algorithm's performance as a black box.
      - "Black box" -- you can run the program but cannot see how it works internally.
- It may even be useful to *deduce* the asymptotic time cost by measuring the time cost for different input sizes.

Procedure for measuring time cost

- Let's suppose we wish to measure the time cost of algorithm *A* as a function of its input size *n*.
- We need to choose the set of values of *n* that we will test.
- If we make *n* too big, our algorithm *A* may never terminate (the input is "too big").
- If we make *n* too small, then *A* may finish so fast that the "elapsed time" is practically 0, and we won't get a reliable clock measurement.

Procedure for measuring time cost

- In practice, one "guesses" a few values for *n*, sees how fast *A* executes on them, and selects a range of values for *n*.
  - Let's define an array of different input sizes, e.g.: `int[] N = { 1000, 2000, 3000, ..., 10000 };`
- Now, for each input size `N[i]`, we want to measure *A*'s time cost.

Procedure for measuring time cost

- Procedure (draft 1):

Make sure to start and stop the clock as “tightly” as possible around the actual algorithm A.

```
for (int i = 0; i < N.length; i++) {
  final Object X = initializeInput(N[i]);

  final long startTime = getClockTime();
  A(X);  // Run algorithm A on input X of size N[i]
  final long endTime = getClockTime();

  final long elapsedTime = endTime - startTime;
  System.out.println("Time for N[" + i + "]: " +
                             elapsedTime);
}
```

Procedure for measuring time cost

- The procedure would work fine if there were no variability in how long `A(X)` took to execute.
- Unfortunately, in the "real world", each measurement of the time cost of `A(X)` is corrupted by *noise*:
  - Garbage collector!
  - Other programs running.
  - Cache locality.
  - Swapping to/from disk.
  - Input/output requests from external devices.

Procedure for measuring time cost

- If we measured the time cost of **`A(X)`** based on *just one measurement*, then our estimate of the "true" time cost *of* **`A(X)`** will be very *imprecise*.
  - We might get unlucky and measure **`A(X)`** while the computer is doing a "system update".
  - If we've very unlucky, this might occur during *some* values of **`i`**, but not for others, thereby *skewing the trend* we seek to discover across the different **`N[i]`**.

Improved procedure for measuring time cost

- A much-improved procedure for measuring the time cost of A(X) is to compute the *average time across M trials*.
- Procedure (draft 2):

```
for (int i = 0; i < N.length; i++) {
  final Object X = initializeInput(N[i]);

  final long[] elapsedTimes = new long[M];
  for (int j = 0; j < M; j++) {
    final long startTime = getClockTime();
    A(X);  // Run algorithm A on input X of size N[i]
    final long endTime = getClockTime();
    elapsedTimes[j] = endTime - startTime;
  }
  final double avgElapsedTime = computeAvg(elapsedTimes);
  System.out.println("Time for N[" + i + "]: " +
                             avgElapsedTime);
}
```

Improved procedure for measuring time cost

- If the elapsed time measured in the *j*th trial is T_j, then the average over all M trials is:

$$\overline{T} = \frac{1}{M}\sum_{j=1}^{M} T_j$$

- We will use the *average time* "*T*-bar" as an estimate of the "true" time cost of A(X).
- The more trials M we use to compute the average, the more precise our estimate "*T*-bar" will be.

Improved procedure for measuring time cost

- Alternatively, we can start/stop the clock just *once*.
- Procedure (draft 2b):

```
for (int i = 0; i < N.length; i++) {
  final Object X = initializeInput(N[i]);

  final long startTime = getClockTime();
  for (int j = 0; j < M; j++) {
    A(X);  // Run algorithm A on input X of size N[i]
  }
  final long endTime = getClockTime();

  final double avgElapsedTime = (endTime - startTime) / M;
  System.out.println("Time for N[" + i + "]: " +
                     avgElapsedTime);
}
```

Quantifying uncertainty

- A key issue in any experiment is to *quantify the uncertainty* of all measurements.
- Example:
  - We are attempting to estimate the "true" time cost of A(X) by averaging together the results of many trials.
  - After computing "T-bar", how far from the "true" time cost of A(X) was our estimate?

Quantifying uncertainty

- A key issue in any experiment is to *quantify the uncertainty* of all measurements.
- Example:
  - You are attempting to estimate the "true" time cost of A(X) by averaging together the results of many trials.
  - After computing "T-bar", how far from the "true" time cost of A(X) was your estimate?
    - In order to compute this, we would have to know what the true time cost is -- and that's what we're trying to estimate!
    - We must find another way to quantify uncertainty...

Standard error versus standard deviation

- Some of you may already be familiar with the *standard deviation*:

$$\sigma = \sqrt{\frac{1}{M}\sum_{j=1}^{M}(T_j - \overline{T})^2}$$

- The standard deviation measures how "varied" the individual measurements T_j are.
  - The standard deviation gives a sense of "how much noise there is."
  - However, in most cases, we are less interested in characterizing the *noise*, and more interested in measuring the *true time cost* of **`A(X)`** itself.
    - For this, we want the *standard error*.

Quantifying your uncertainty

- In statistics, the uncertainty associated with a measurement (e.g., the time cost of A(X)) is typically quantified using the *standard error*:

Standard deviation

$$\mathrm{StdErr} = \frac{\sigma}{\sqrt{M}} \qquad \text{where} \qquad \sigma = \sqrt{\frac{1}{M}\sum_{j=1}^{M}(T_j - \overline{T})^2}$$

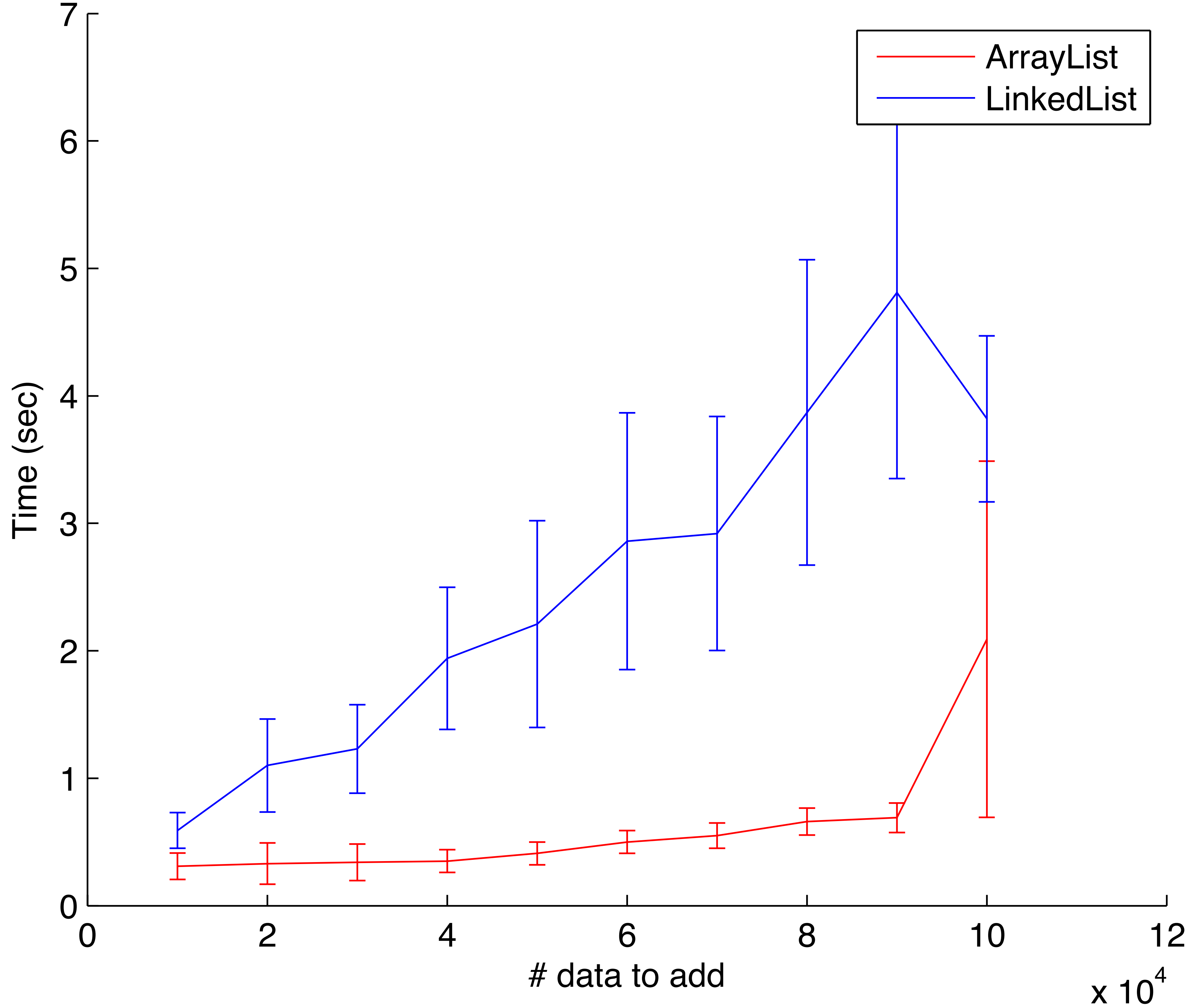
where "T-bar" is the average (computed on earlier slide).

- Notice: as *M* grows larger, the StdErr becomes smaller.

Error bars

- The standard error is often used to compute *error bars* on graphs to indicate how reliable they are.
  - Different error bars have different meanings! Some of them indicate confidence intervals, some indicate standard error, some indicate standard deviation -- it's important to know which!

Example


ArrayList
LinkedList
Time (sec)
0
1
2
3
4
5
6
7
0
2
4
6
8
10
12
data to add
x 10^4

Stacks and queues.

Stacks and queues.

- Let's now bring in two more fundamental data structures into the course.
- So far we have covered lists -- array-based lists and linked-lists.
  - These are both linear data structures -- each element in the container has at most one *successor* and one *predecessor*.
- Lists are most frequently used when we wish to store objects in a container, and *probably never remove them from it*.
  - E.g., if Amazon uses a list to store its huge collection of customers, it has no intention of "removing" a customer (except at program termination).

Stacks and queues

- Stacks and queues, on the other hand, are examples of *linear* data structures in which every object inserted into it will generally be removed:
  - The stack/queue is intended only as “temporary” storage.
- Both stacks and queues allow the user to add and remove elements.
- Where they differ is the *order* in which elements are removed *relative to when they were added*.

Stacks

- Stacks are *last-in-first-out* (LIFO) data structures.
- The classic analogy for a "stack" is a pile of dishes:
  - Suppose you've already added dishes A, B, and C to the "stack" of dishes.
  - Now you add one more, D.
  - Now you remove one dish -- *you get D back*.
  - If you remove another, you get C, and so on.
- With stacks, you can only add to/remove from the *top* of the stack.

If you try to remove a middle dish, you get that annoying clanging sound.

Stacks

- Stacks find many uses in computer science, e.g.:
  - Implementing *procedure calls*.
- Consider the following code:

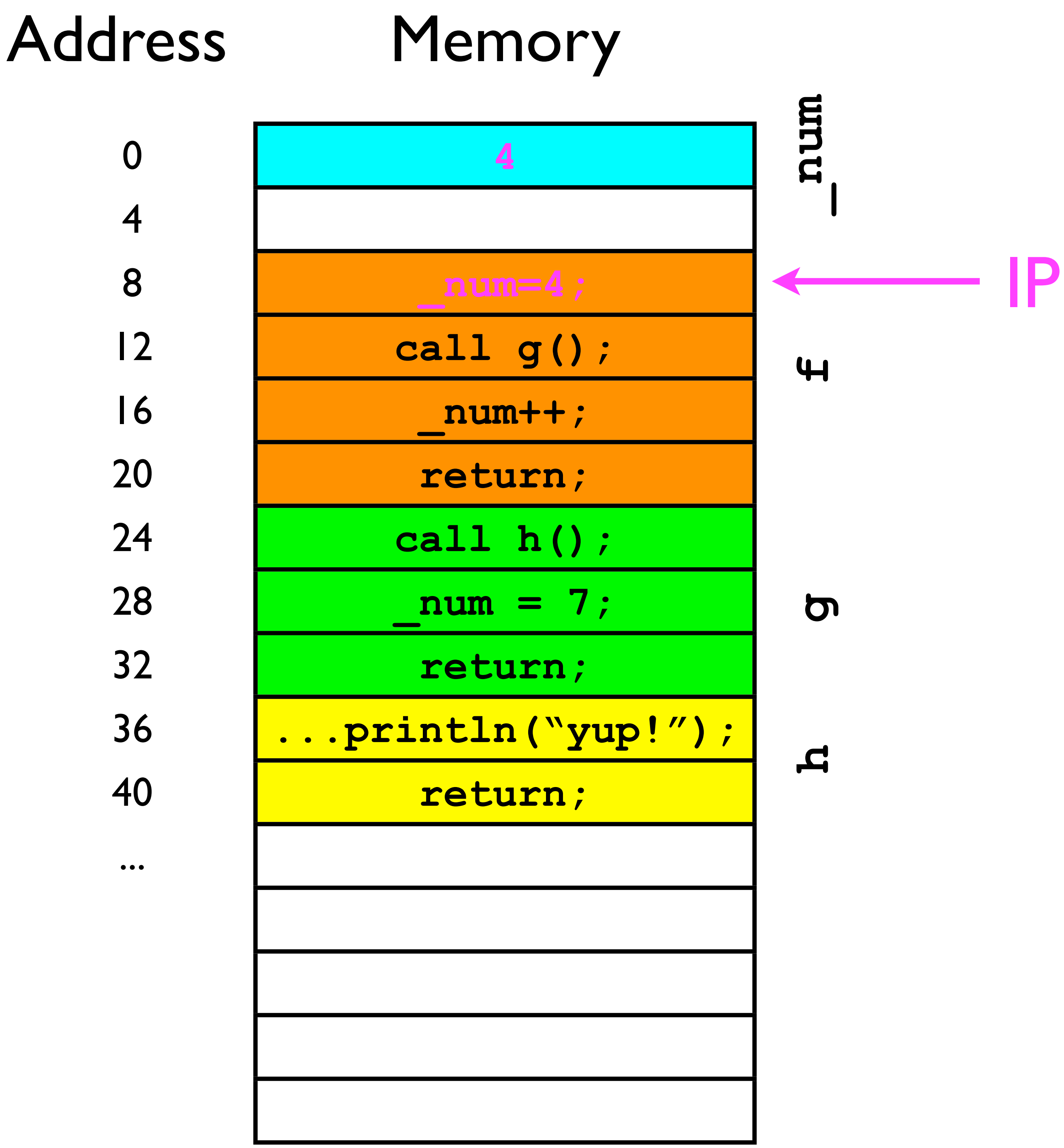
```
void f () {
  _num = 4;
  g();
  _num++;
}
void g () {
  h();
  _num = 7;
}
void h () {
  System.out.println("Yup!");
}
```

How does the CPU know to "jump" from `f` to `g`, `g` to `h`, then `h` back to `g`, and finally `g` back to `f`?

Von Neumann machine

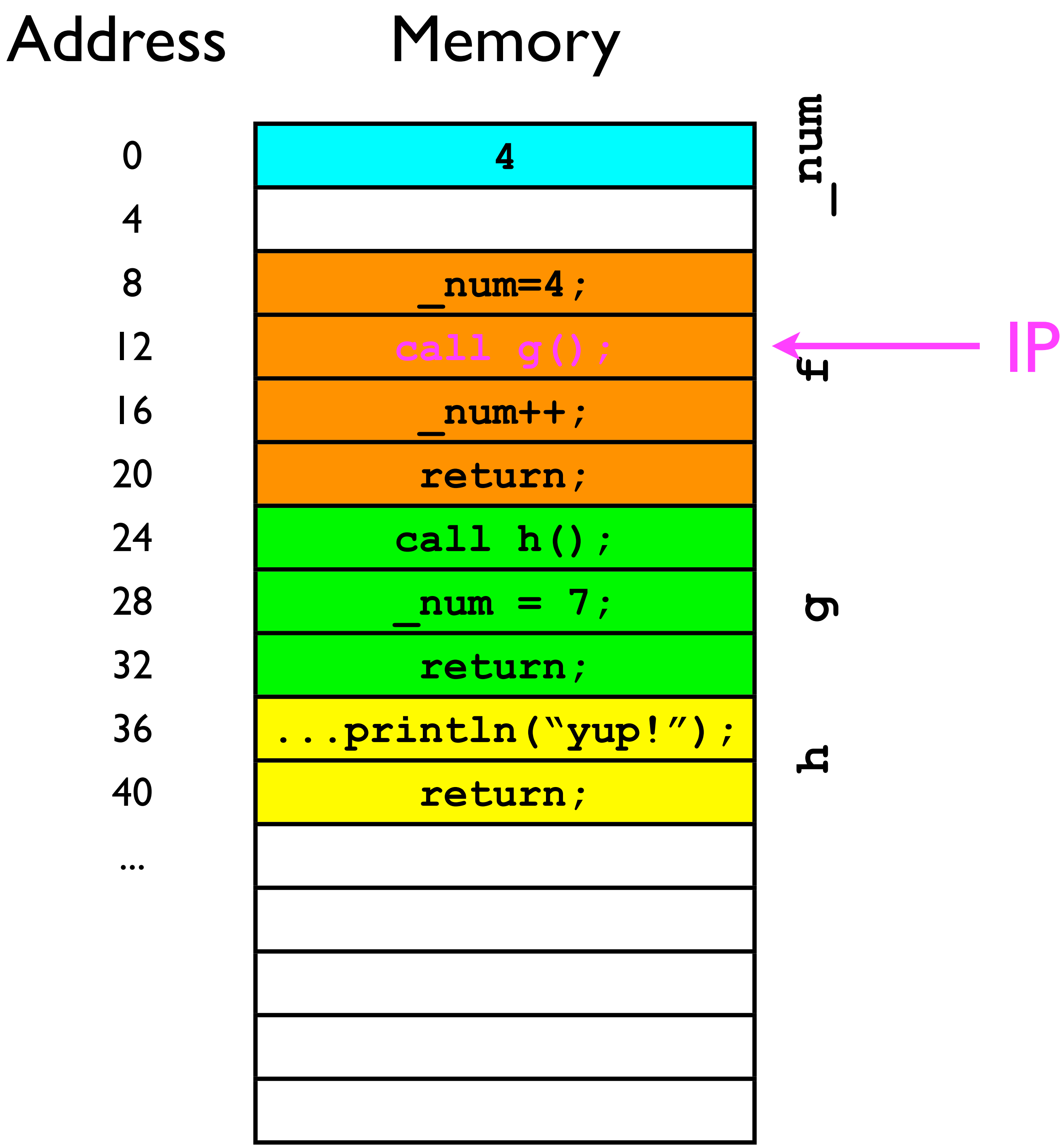
- On all modern machines, a program's *instructions* and its *data* are stored *together* somewhere in the computer's long sequence of bits (Von Neumann architecture).
    - Just by "glancing" at the contents of computer memory, one would have no idea whether a certain byte contains code or data -- it's all just bits.
- To keep track of which instruction in memory is currently being executed, the CPU maintains an Instruction Pointer (IP).

Code execution



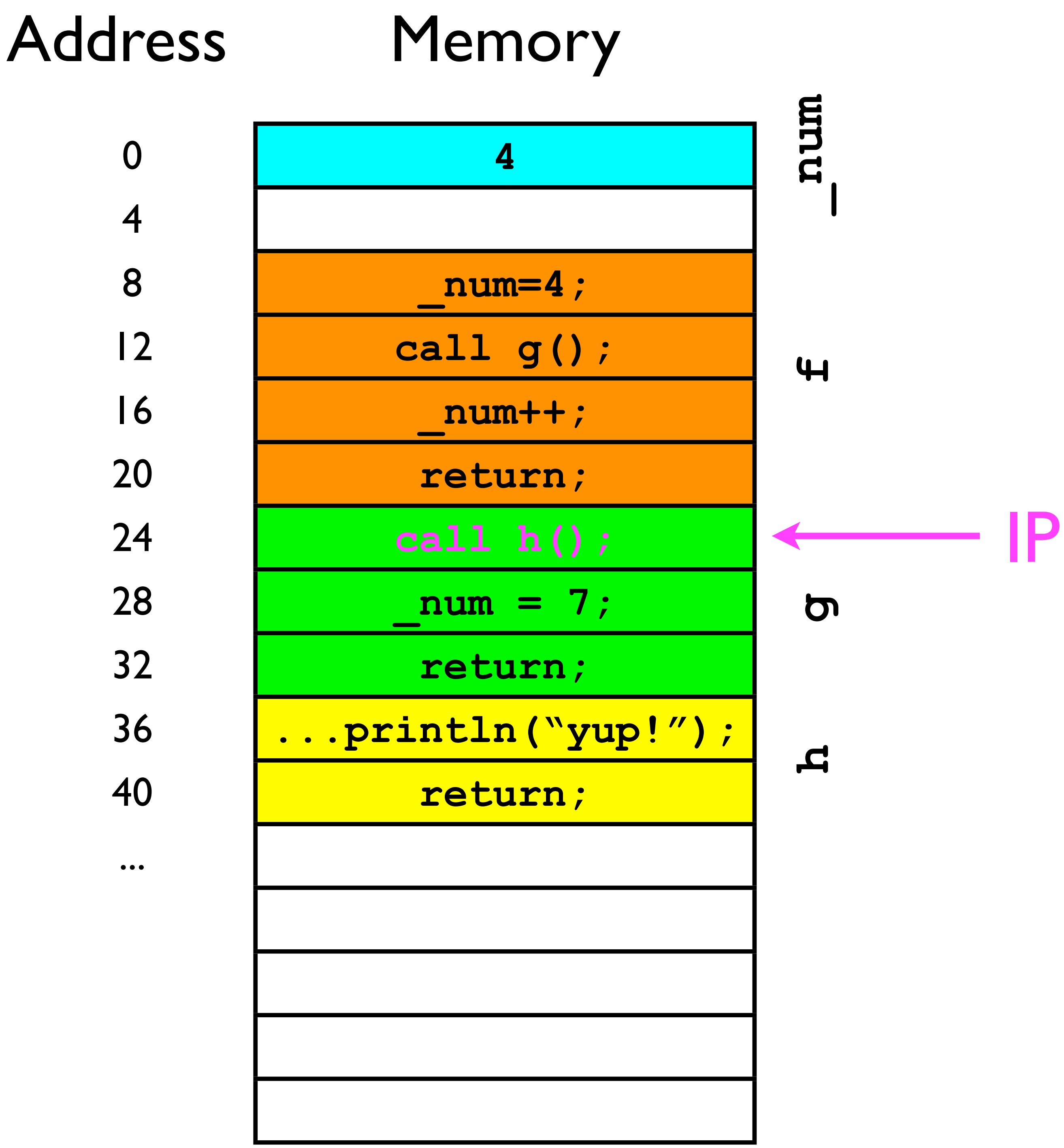
- Suppose the IP is 8:
  - Then the next instruction to execute is **`_num=4`**;
- The CPU then advances the IP to the next instruction (4 bytes later) to 12.

Code execution



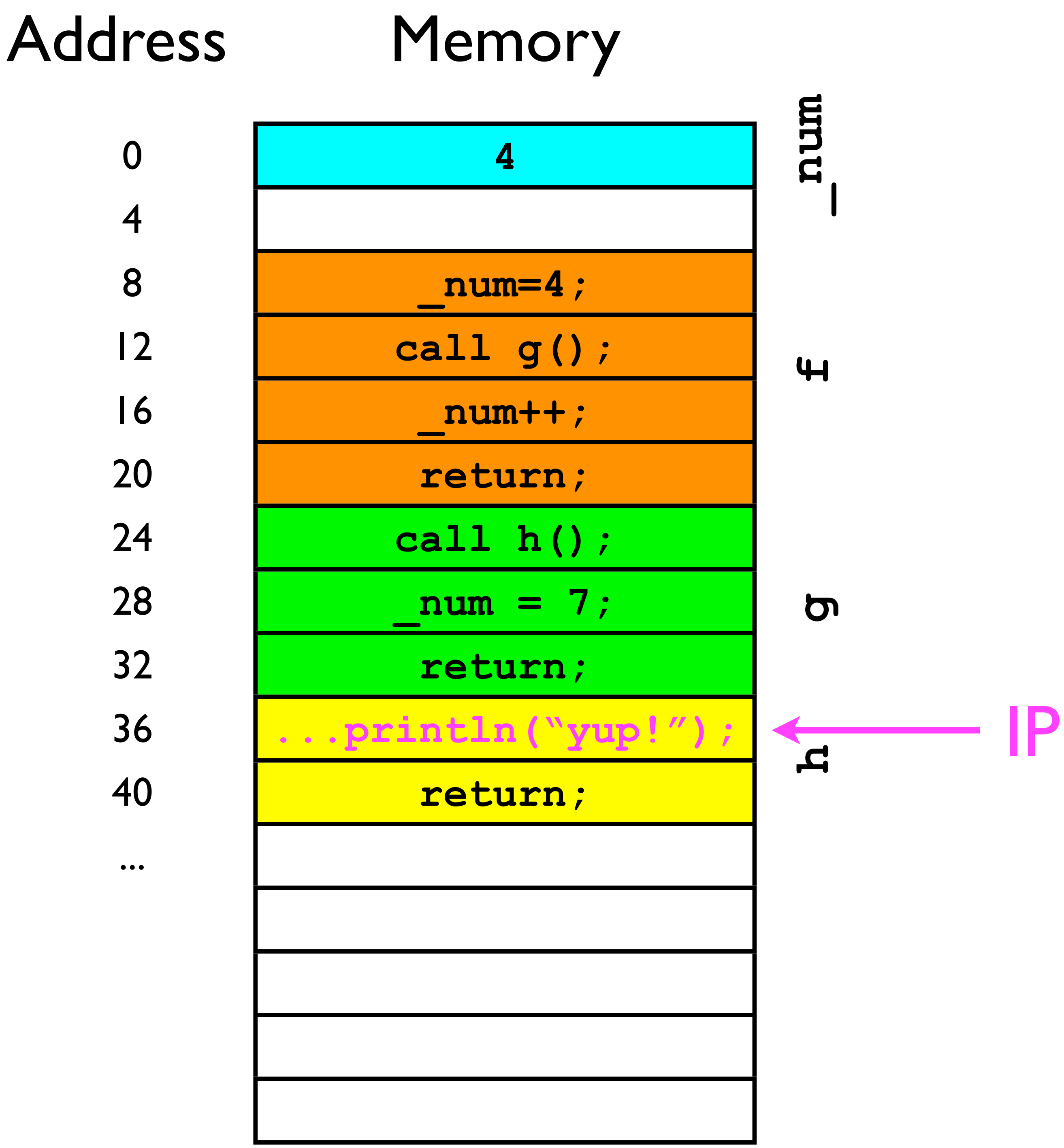
- The next instruction is `call g()`.
- The CPU must now "move" the IP to address 24 (start of g's code) so **g** can start.

Code execution



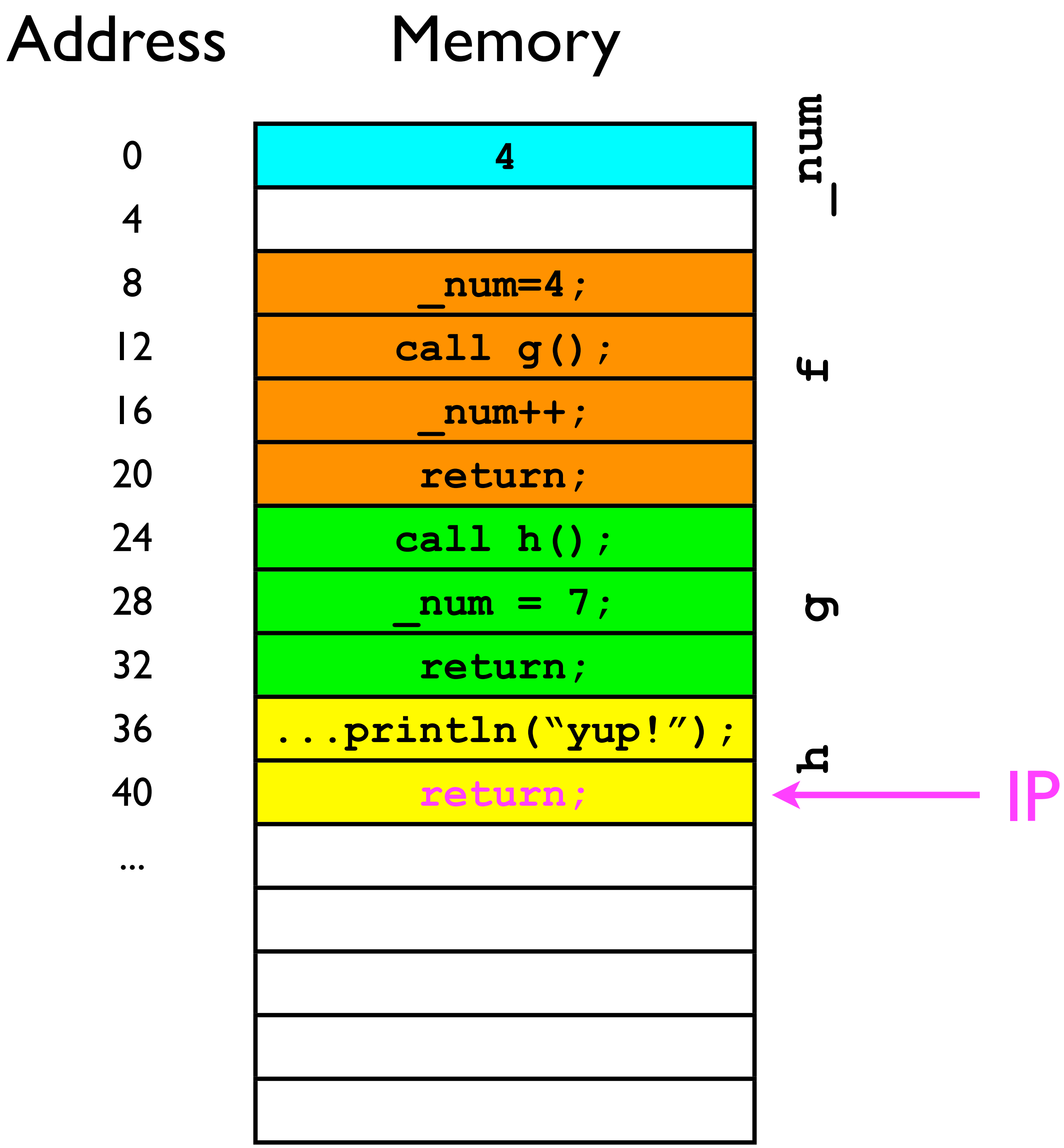
- **g** has now started.
- The first thing **g** does is call **h**.
- We have to move the IP again.

Code execution



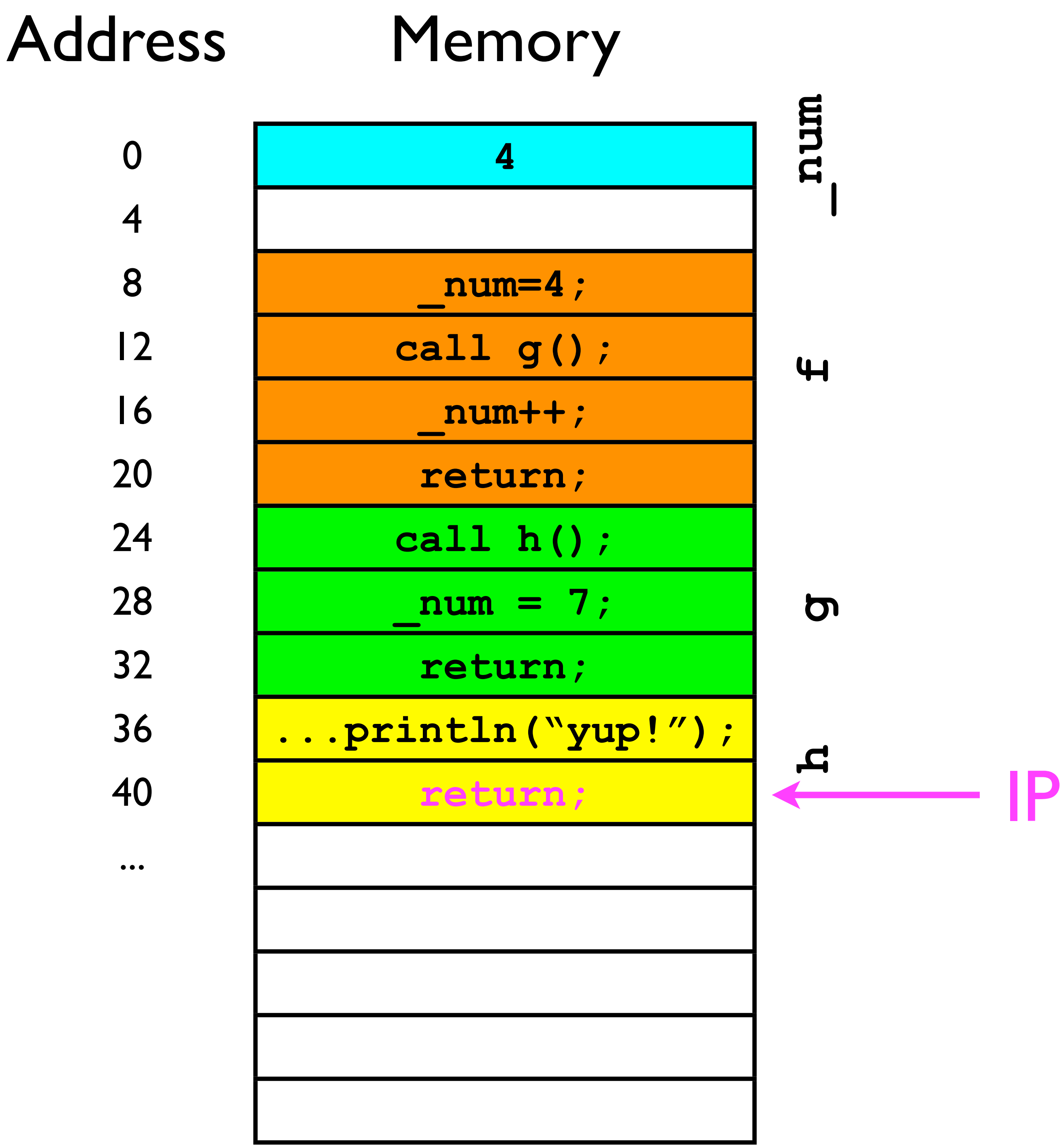
- **h** now prints out "yup!".

Code execution



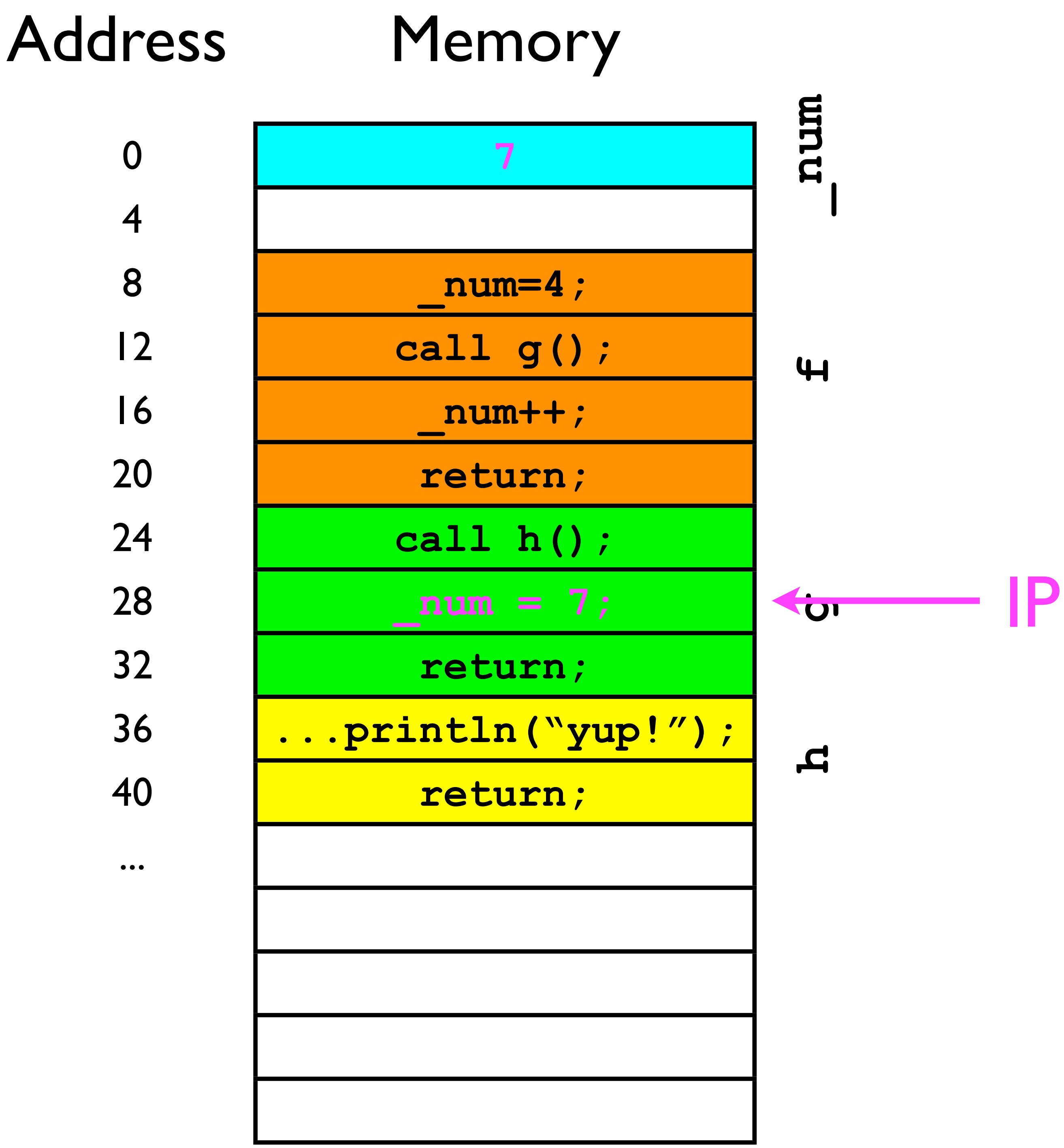
- The return instructions tells the CPU to move the IP back to where it was *before the current method was called.*
- But where is that?

Code execution



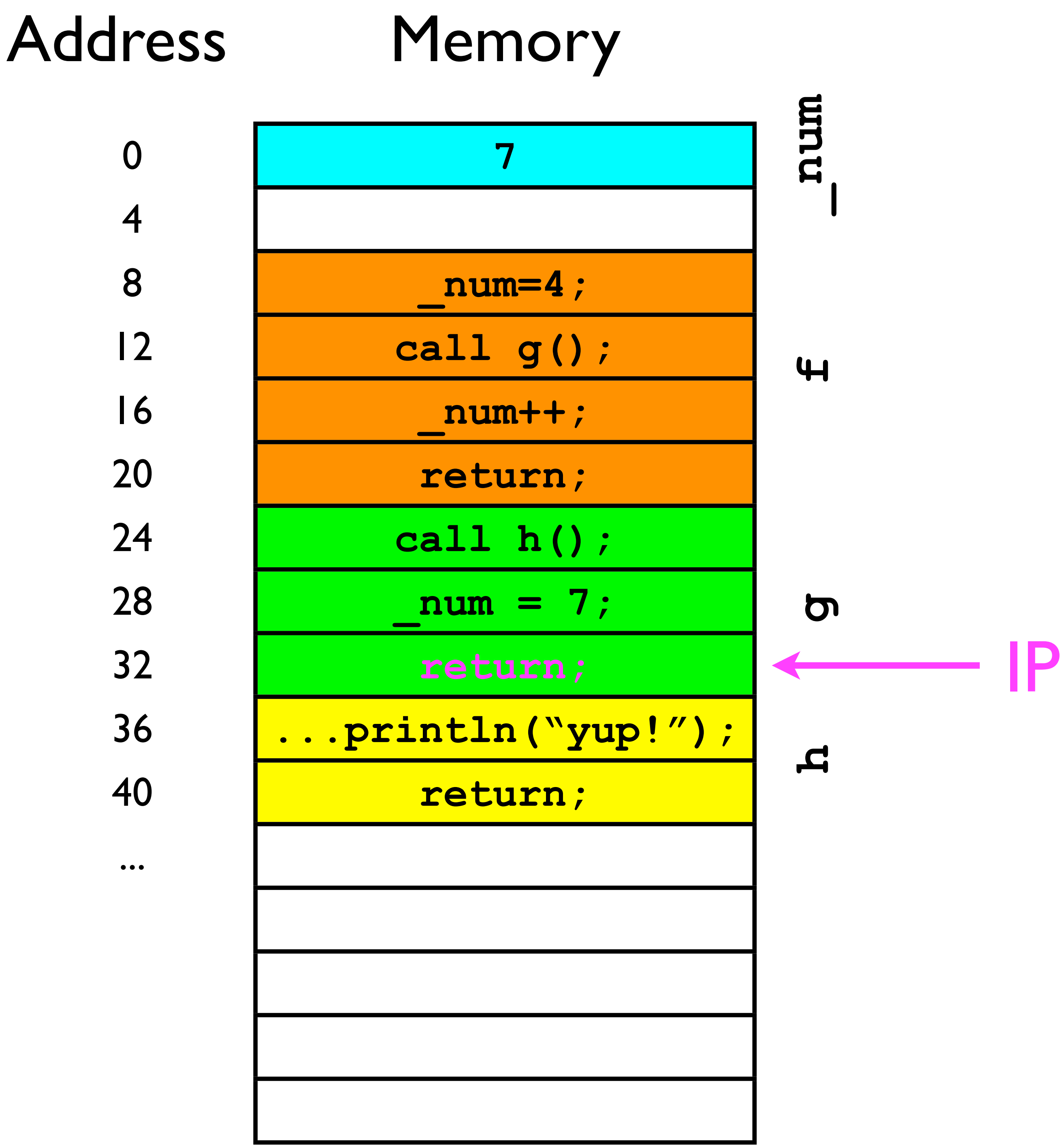
- The return call at address 40 *should* cause the CPU to jump to address 28 -- *the next instruction in* **g**.

Code execution



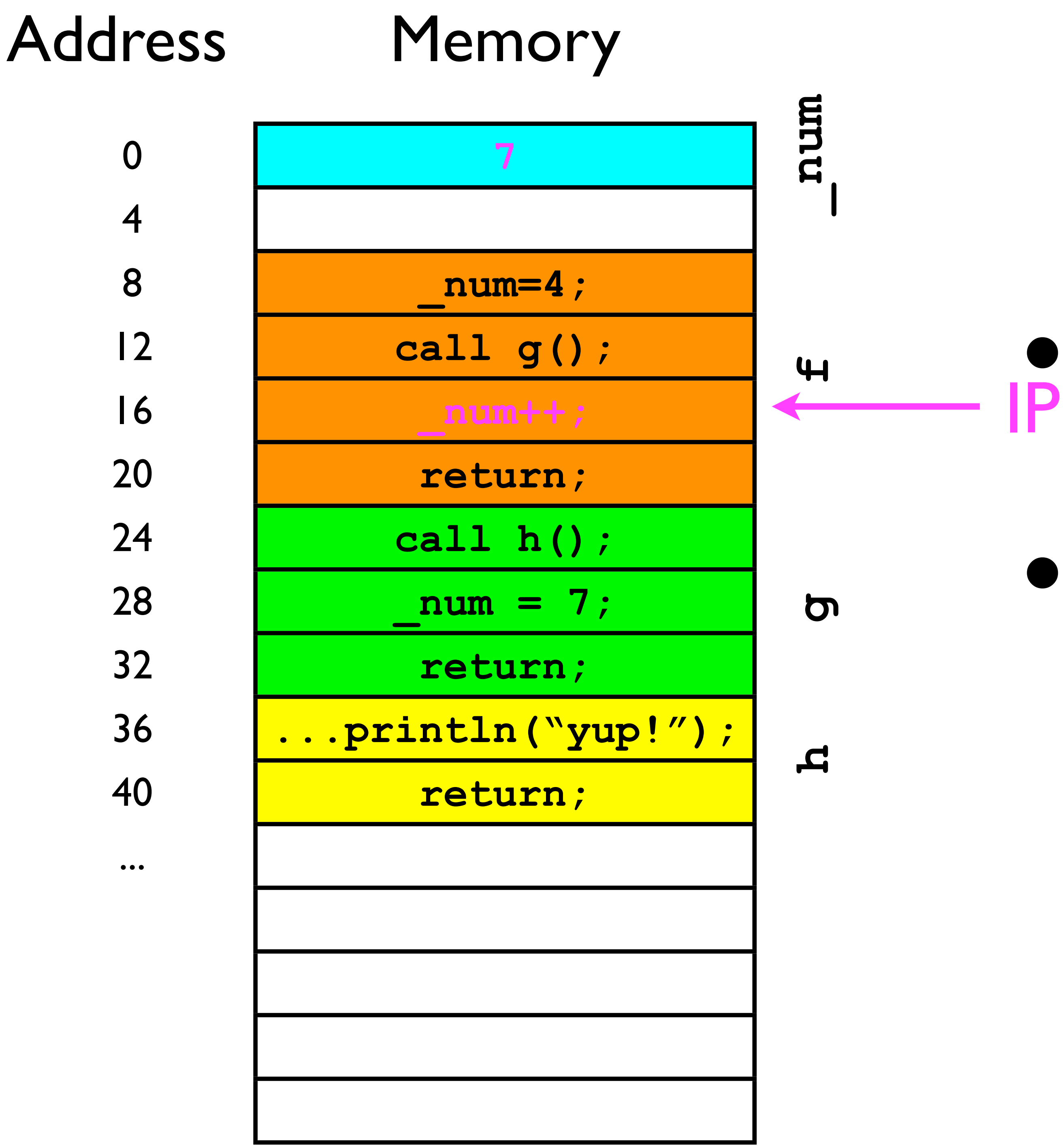
- We then execute `_num=7;`

Code execution



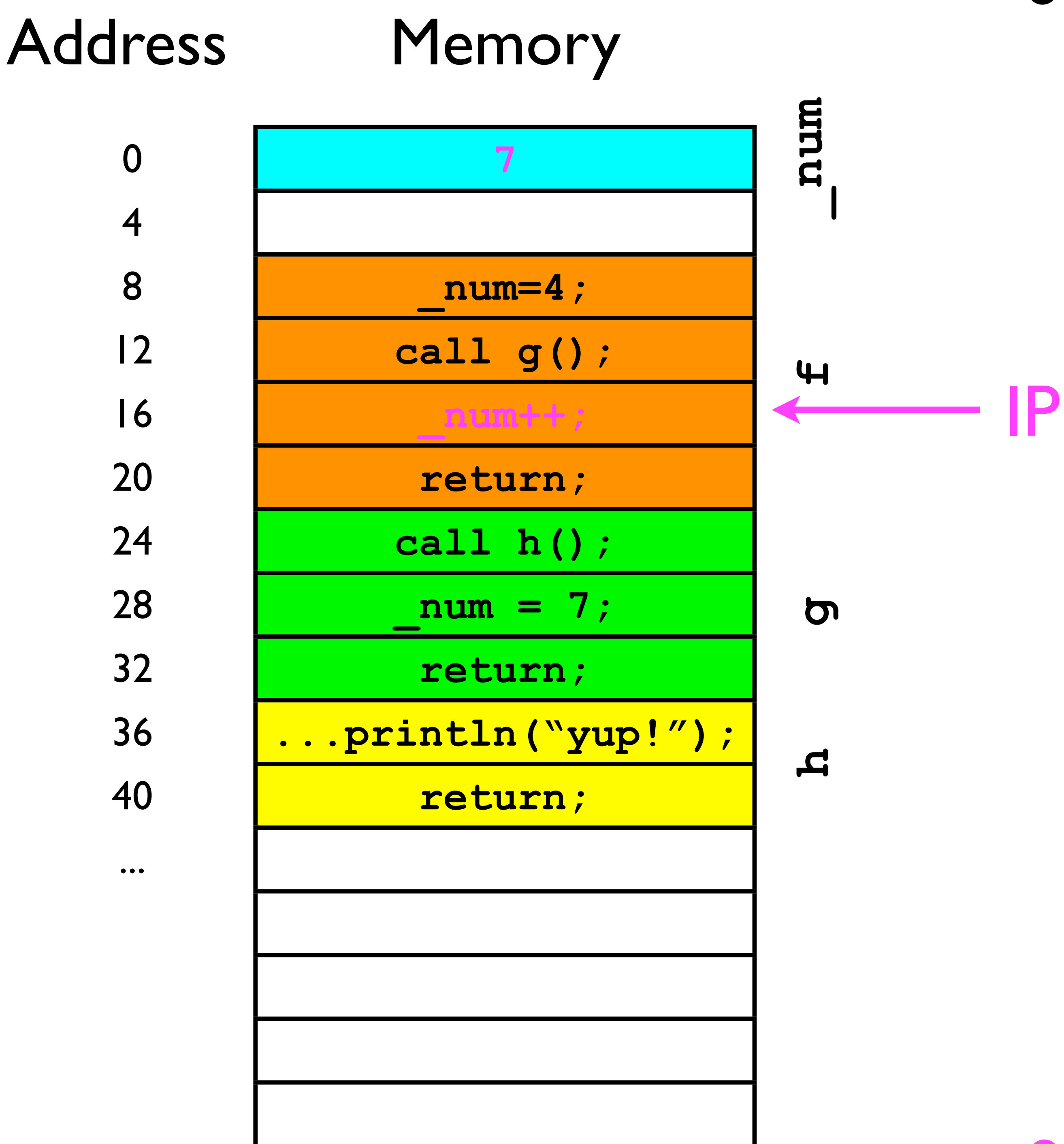
- And now we have to return to where the *caller* of g left off (address 16).

Code execution



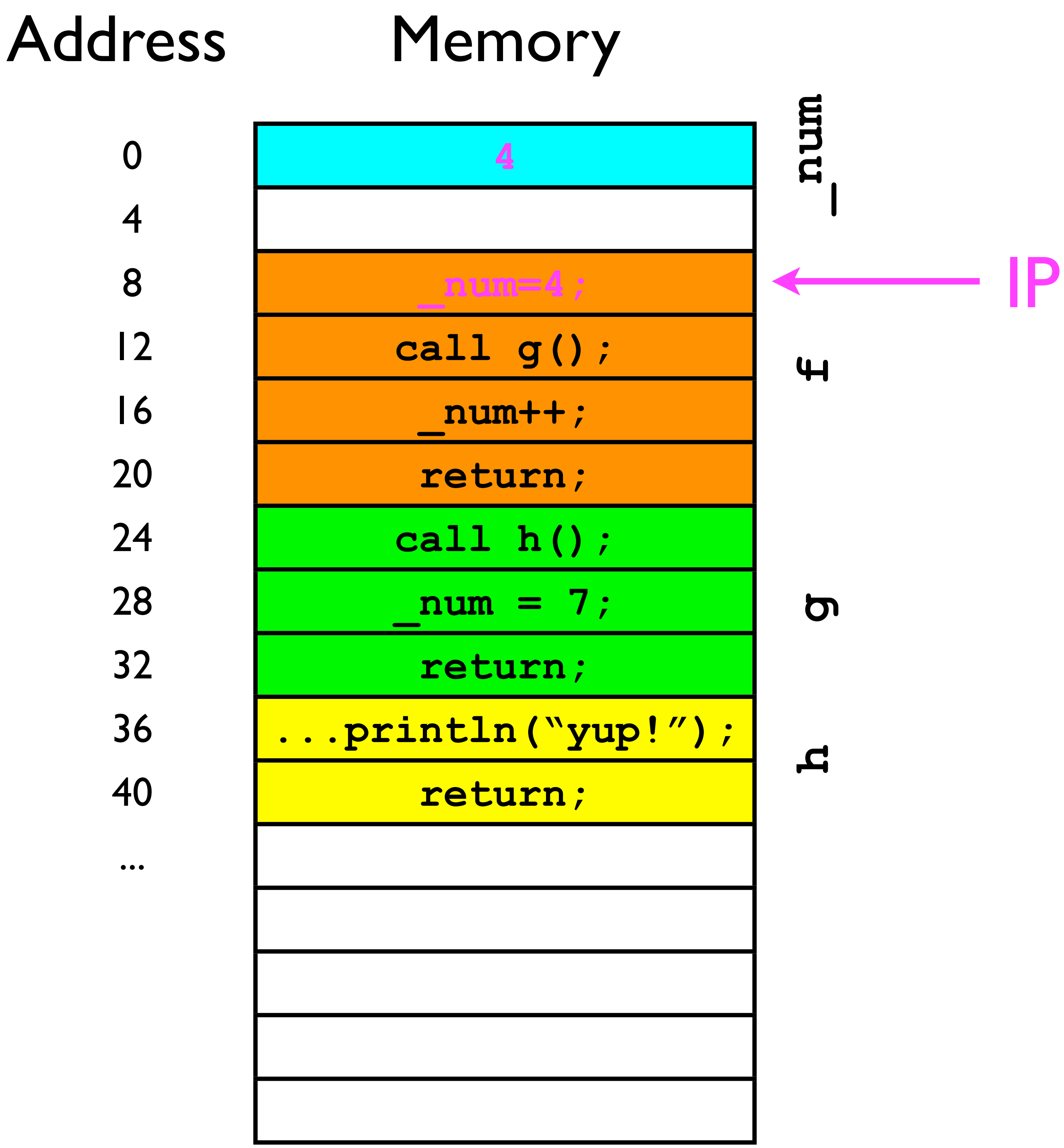
- How does the CPU know which address to "return" to?
- We need some kind of data structure to manage the "return addresses" for us.

Code execution



- What we need is a last-in-first-out data structure ("stack") to remember all the return addresses:
  - *Rule 1*: Before method x calls method y, method x first adds its "return address" to the stack.
  - *Rule 2*: When method y "returns" to its caller, it removes the top of the stack and sets the IP to that address.
- Let's see this work in practice...

Code execution



- "Return address" stack:

(bottom of stack)

Code execution

Address	Memory	
0	4	_num
4		
8	_num=4;	f
12	call g(); ← IP	f
16	_num++;	f
20	return;	f
24	call h();	g
28	_num = 7;	g
32	return;	g
36	...println("yup!");	h
40	return;	h
...		

- "Return address" stack:

"push" 16 onto stack

16

(bottom of stack)

Code execution

Address	Memory	
0	`4`	`_num`
4		
8	`_num=4;`	`f`
12	`call g();`	`f`
16	`_num++;`	`f`
20	`return;`	`f`
24	`call h();` ← IP	`g`
28	`_num = 7;`	`g`
32	`return;`	`g`
36	`...println("yup!");`	`h`
40	`return;`	`h`
...		

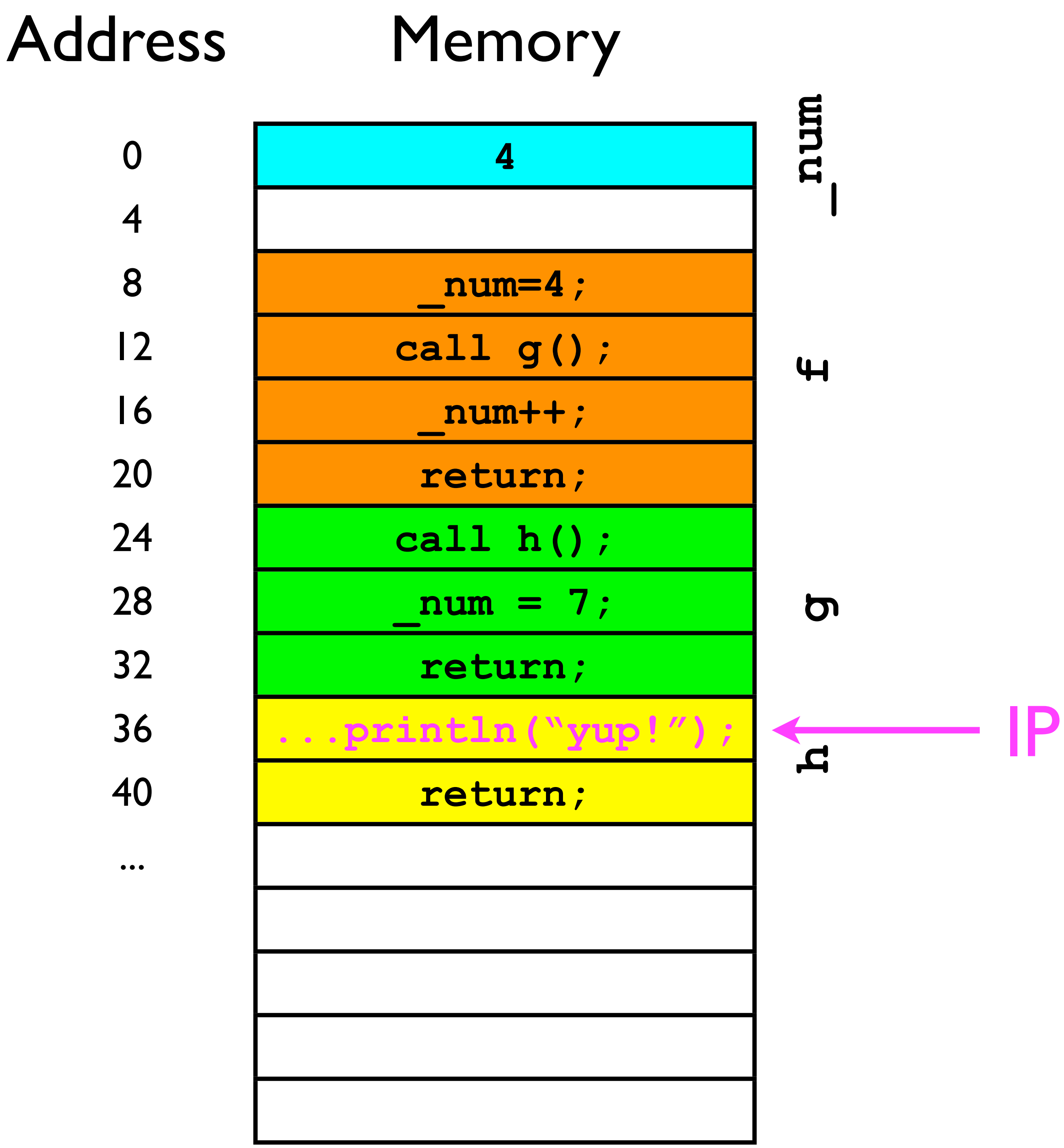
- "Return address" stack:

"push" 28 onto stack

28
16

(bottom of stack)

Code execution



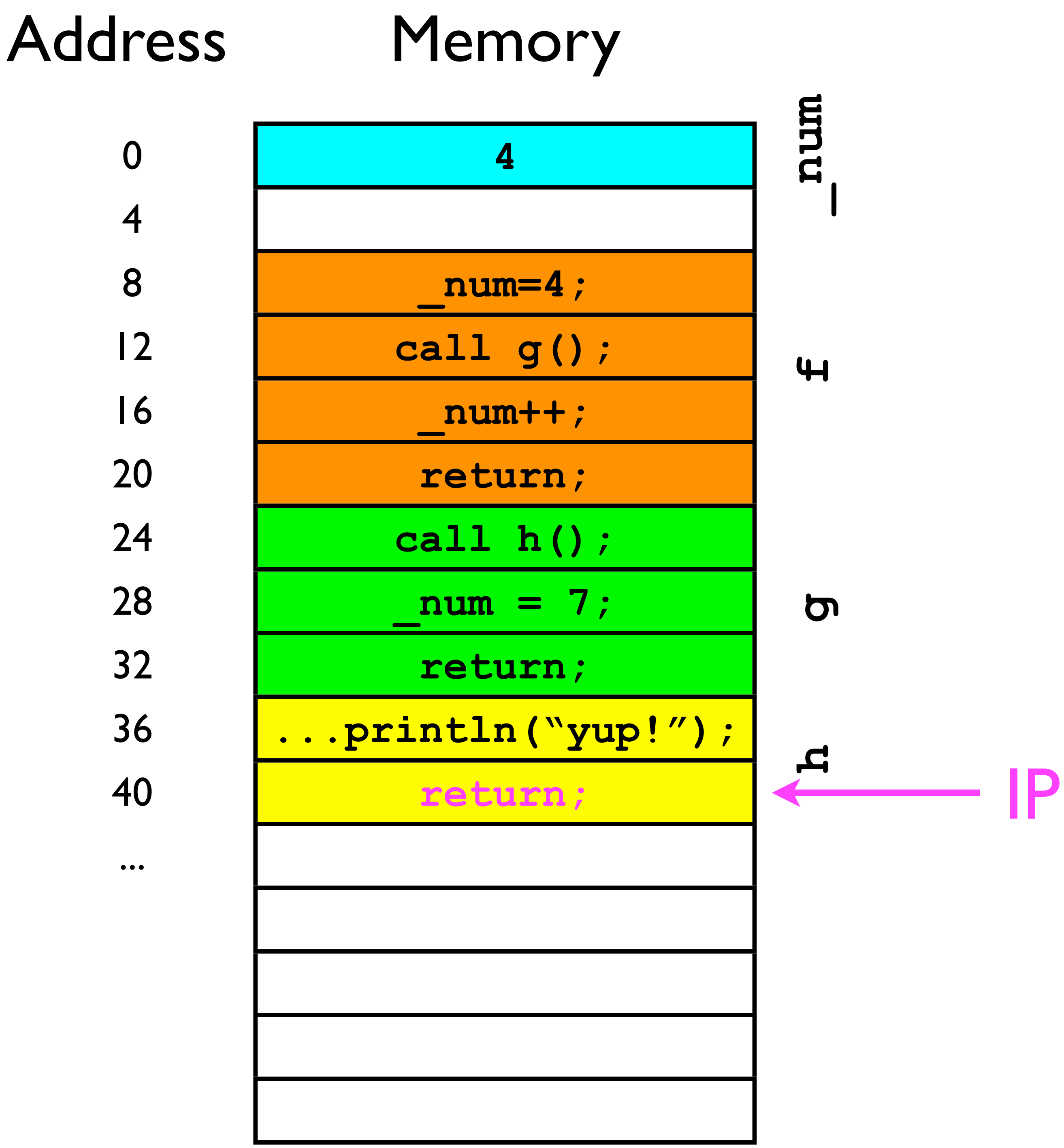
- "Return address" stack:

28

16

(bottom of stack)

Code execution



- "Return address" stack:

"pop" 28 off the stack...

28
16

(bottom of stack)

Code execution

Address	Memory		
0	7	`_num`	
4			
8	`_num=4;`		
12	`call g();`	`f`	
16	`_num++;`		
20	`return;`		
24	`call h();`		
28	`_num = 7;`	`g`	IP
32	`return;`		
36	`...println("yup!");`	`h`	
40	`return;`		
...			

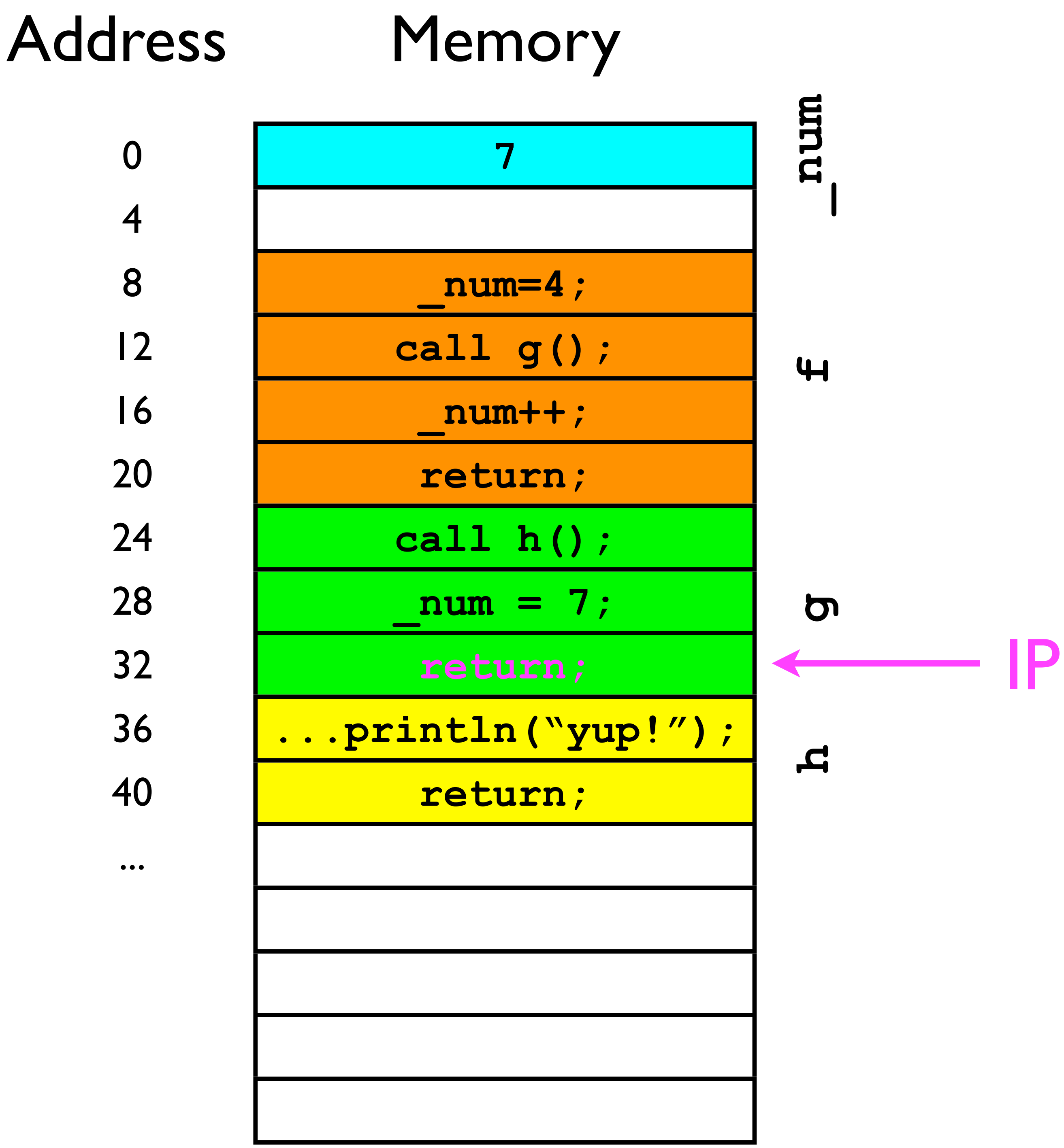
- "Return address" stack:

...and jump to that address.

16

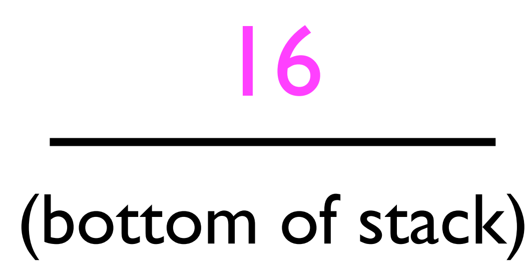
(bottom of stack)

Code execution

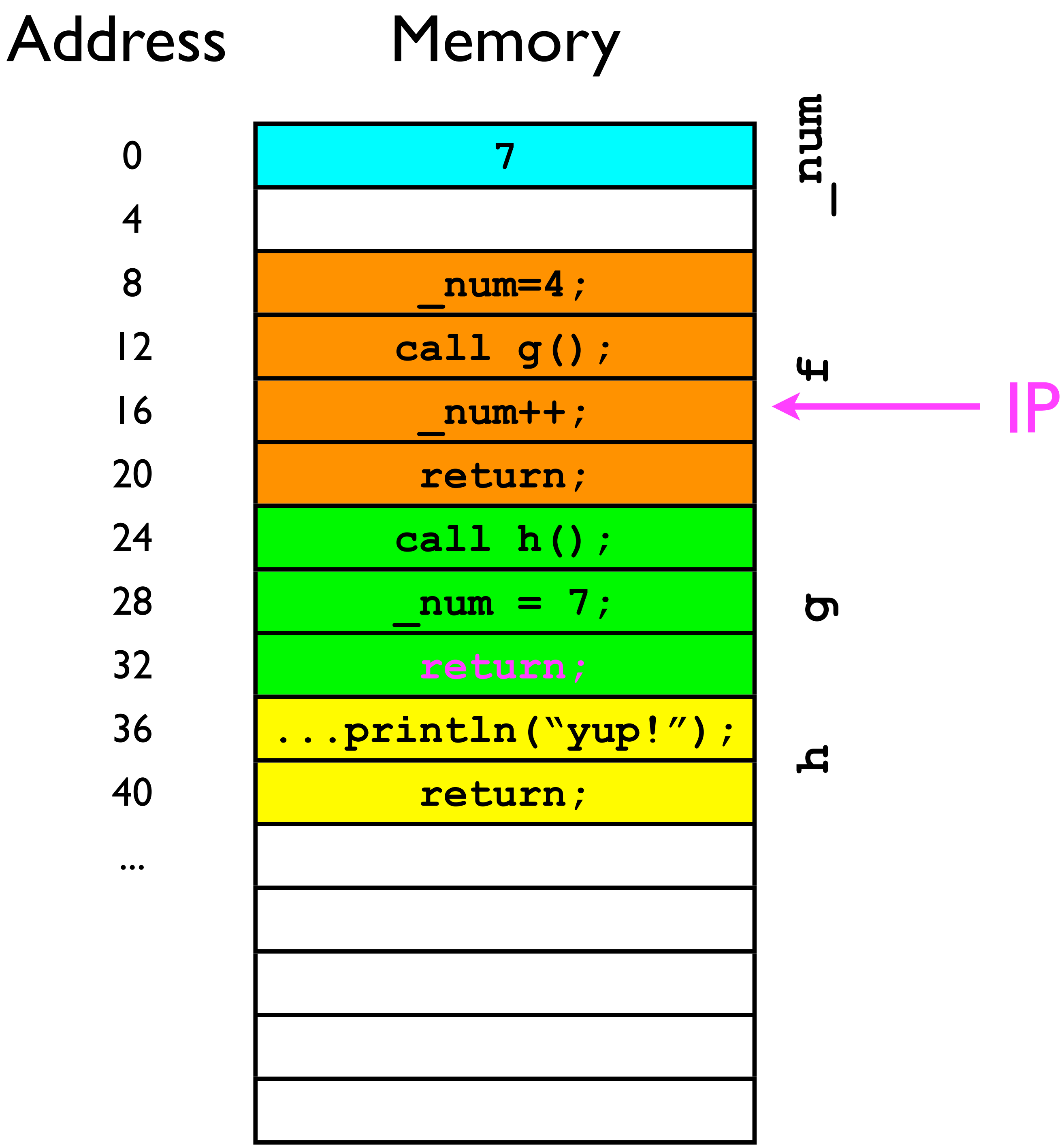


- "Return address" stack:

"pop" 16 off the stack...



Code execution



- "Return address" stack:

...and jump to that address.

(bottom of stack)

Stack ADT

- To support the last-in-first-out adding/removal of elements, a stack must adhere to the following interface:

```
interface Stack<T> {
  // Adds the specified object to the top of the stack.
  void push (T o);

  // Removes the top of the stack and returns it.
  T pop ();

  // Returns the top of the stack without removing it.
  T peek ();
}
```

Stack ADT

- Similarly to a *list*, a *stack* can be implemented straightforwardly using two kinds of backing stores:
  - Linked list
  - Array
- Think about how these would work...
  - In the case of linked list, our StackImpl class might start out like:

```
class StackImpl<T> {
  DoublyLinkedList12<T> _underlyingStorage;
}
```