Iteration

Rob Hackman

Fall 2025

University of Alberta

Table of Contents

Repetition

When we have needed to run code repeatedly an arbitrary number of times we have relied on recursion as our tool for repetition.

Repetition

When we have needed to run code repeatedly an arbitrary number of times we have relied on recursion as our tool for repetition.

However, while recursion in addition to the rest of the language constructs we've learned is sufficient to solve any problem that is computable. So while new forms of repetition aren't *necessary* they can be useful.

New forms of repetition

In computing there are two main forms of repetition, the first being recursion (and by extension our higher-order functions such as folds).

The other form is known as *iteration*. Despite the fact that the English word iterate simply means to repeat, in computing when we use the word iterate we are usually referring to the particular constructs we are about to learn about, rather than recursion. This mechanism we are about to learn about are also referred to as *loops*.

New Python keyword `while`

The first language construct that supports iteration we will learn about is the `while` loop.

New Python keyword `while`

The first language construct that supports iteration we will learn about is the `while` loop.

The `while` keyword is used to write a compound statement which takes the form:

New Python keyword `while`

The first language construct that supports iteration we will learn about is the `while` loop.

The `while` keyword is used to write a compound statement which takes the form:

```
while expr:
  statement_1
  statement_2
  ...
  statement_n
```

`while` loops

When a `while` statement is reached by the Python interpreter it does the following steps, which is very similar to an if conditional.

1. Evaluate the expression
2. If the result of evaluating the expression is False, go to the code after the `while` statement skipping over the statements within the `while` statement much like an if statement
3. If the expression is True execute the statements within the `while` statement, much like an if statement
    - Where `while` statements differ from if statements is at this point — after all the lines within the `while` statement have been executed rather than continuing on to the statements after execution continues on back to step one listed above and the expression is evaluated again

Structuring a `while` loop

When writing a `while` there are four different abstract components a programmer should think about, every `while` loop can really be thought of as the following format

```
LoopInitialization
while LoopCond:
  LoopBody
  LoopUpdate
```

Structuring a `while` loop

```
LoopInitialization
while LoopCond:
  LoopBody
  LoopUpdate
```

- Loop Initialization — this is code that comes *before* the loop which sets up the values stored in identifiers to our initial values

Structuring a `while` loop

```
LoopInitialization
while LoopCond:
  LoopBody
  LoopUpdate
```

- Loop Initialization — this is code that comes *before* the loop which sets up the values stored in identifiers to our initial values
- Loop Condition — this is the expression that will determine when our loop ends, if this condition is never False then we have an *infinite loop*

Structuring a `while` loop

```
LoopInitialization
while LoopCond:
  LoopBody
  LoopUpdate
```

- Loop Initialization — this is code that comes *before* the loop which sets up the values stored in identifiers to our initial values
- Loop Condition — this is the expression that will determine when our loop ends, if this condition is never False then we have an *infinite loop*
- Loop Body — this is the content of the loop, the actual function we want to repeat

Structuring a `while` loop

```
LoopInitialization
while LoopCond:
  LoopBody
  LoopUpdate
```

- Loop Initialization — this is code that comes *before* the loop which sets up the values stored in identifiers to our initial values
- Loop Condition — this is the expression that will determine when our loop ends, if this condition is never False then we have an *infinite loop*
- Loop Body — this is the content of the loop, the actual function we want to repeat
- Loop Update — a statement, or series of statements, that updates the variables that are used in our Loop Condition to make sure we are stepping closer to the loop terminating

`while` familiarities

If we think about the four components of our code we must consider when writing a `while` loop they seem oddly familiar...

It becomes quite clear that there are many parallels between iterative and recursive solutions to problems!

`while` familiarities

If we think about the four components of our code we must consider when writing a `while` loop they seem oddly familiar...

- Loop Initialization — Initial arguments provided when calling a recursive function, or base case value

It becomes quite clear that there are many parallels between iterative and recursive solutions to problems!

`while` familiarities

If we think about the four components of our code we must consider when writing a `while` loop they seem oddly familiar...

- Loop Initialization — Initial arguments provided when calling a recursive function, or base case value
- Loop Condition — Base case condition

It becomes quite clear that there are many parallels between iterative and recursive solutions to problems!

`while` familiarities

If we think about the four components of our code we must consider when writing a `while` loop they seem oddly familiar...

- Loop Initialization — Initial arguments provided when calling a recursive function, or base case value
- Loop Condition — Base case condition
- Loop Body — Recursive case code

It becomes quite clear that there are many parallels between iterative and recursive solutions to problems!

`while` familiarities

If we think about the four components of our code we must consider when writing a `while` loop they seem oddly familiar...

- Loop Initialization — Initial arguments provided when calling a recursive function, or base case value
- Loop Condition — Base case condition
- Loop Body — Recursive case code
- Loop Update — How we step closer to the base case

It becomes quite clear that there are many parallels between iterative and recursive solutions to problems!

Using `while` loops

Lets now write a function without using recursion or higher-order functions, only use `while` loops when we need to repeat.

We will start with a simple function, the function that returns the sum of all the integers in a `LList` of ints.

```
def llistSum(l):
```

Using `while` loops

To begin writing our loop we must start by writing our loop initialization. Unlike our base case we don't immediately know the answer to a question, so we must consider what data we're going to need in our loop and how we will use it.

Since we are trying to calculate a sum we will need a variable in which to keep the total sum — that variable must be initialized to something. We don't want that initial value to affect the sum of items in our `LList` so the additive identity zero is a good choice!

```
def llistSum(l):
  resSoFar = 0
```

Using `while` loops

Next we must consider our loop condition — when are we finished doing work?

Whenever we've looked at each item in `l`. Recursively that was when the function was called when `l` is empty, so perhaps that is a good choice for condition here as well. Again, this is not as simple as thinking recursively where we only have to solve the problem for the current state of our data. Now we must also consider how our data is going to change in the future.

```
def llistSum(l):
  resSoFar = 0
  while not isEmpty(l):
```

Using `while` loops

We should now write our loop body. Ultimately we need to add each item of `l` to our result. As always the current item we can access is the first item, so we should do that.

```
def llistSum(l):
  resSoFar = 0
  while not isEmpty(l):
    resSoFar = resSoFar + first(l)
```

Using `while` loops

Lastly we need to consider our loop update statement. Our loop update statement needs to handle two tasks in this case:

- It must modify the state of our variables so that we are closer to the loop ending
- It must modify the state of our variables so that should our loop run again it does the next step of our work

Luckily in that case both of these are achieved by the same thing — we must reassign `l` to be the `rest` of `l`.

```
def llistSum(l):
  resSoFar = 0
  while not isEmpty(l):
    resSoFar = resSoFar + first(l)
    l = rest(l)
```

Using `while` loops

Finally after our loop is complete our answer should be computed, and we mustn't forget to return it!

In this case we want to make sure the `return` statement is *after* the loop and not inside it, as if it was inside it our function would stop running after the first time the loop runs!

```
def llistSum(l):
  resSoFar = 0
  while not isEmpty(l):
    resSoFar = resSoFar + first(l)
    l = rest(l)
  return resSoFar
```

Comparing `llistSum`

Consider the two version of `llistSum` below.

```
def llistSum(l):
  resSoFar = 0
  while not isEmpty(l):
    resSoFar = resSoFar + first(l)
    l = rest(l)
  return resSoFar
```

```
def llistSum(l):
  if isEmpty(l):
    return 0
  ror = llistSum(rest(l))
  return first(l) + ror
```

The commonalities between these two functions, and how they've been designed, is easy to see.

Tracing loops

Trying to understand code that is executed in a loop is more difficult than code recursive solutions.

In our recursive solutions each line executed once for each function application, and we could easily expand each function call into its resultant expression until we were left with a final non-recursive expression we could calculate.

In our iterative solution the lines within the loop are executed an arbitrary number of times. We also cannot say what the result of executing a line of code will be without considering the current state of our program — so to follow the code executed by a loop we must track the values of all our variables as the loop executes.

Tracing a `while` loop

```
def llistSum(l):
  resSoFar = 0
  while not isEmpty(l):
    resSoFar = resSoFar + first(l)
    l = rest(l)
  return resSoFar
```
[1]

Let us now trace the function call `llistSum(LL(7, 22, 3, 9))`.

To perform such a trace we must keep track of the state of all our variables as we execute our code.

Tracing a `while` loop

```
1  def llistSum(l):
2    resSoFar = 0
3    while not isEmpty(l):
4      resSoFar = resSoFar + first(l)
5      l = rest(l)
6    return resSoFar
```

```
l : LL(7, 2, 3, 9)
line to exec : 2
```

First the function is called, and our initial state is shown to the right. We will also track in our state the next line to be executed.

Tracing a `while` loop

```
1  def llistSum(l):
2    resSoFar = 0
3    while not isEmpty(l):
4      resSoFar = resSoFar + first(l)
5      l = rest(l)
6    return resSoFar
```

```
l : LL(7, 2, 3, 9)
resSoFar : 0
line to exec : 3
```

After line 2 is executed our state is updated with `resSoFar` being initialized.

Tracing a `while` loop

```
1  def llistSum(l):
2    resSoFar = 0
3    while not isEmpty(l):
4      resSoFar = resSoFar + first(l)
5      l = rest(l)
6    return resSoFar
```

```
l : LL(7, 2, 3, 9)
resSoFar : 0
line to exec : 4
```

Line 3 is executed and the condition `not isEmpty(l)` is evaluated. Since L is not empty we enter the code within the `while` statement, meaning line 4 is the next line to execute.

Tracing a `while` loop

```
1  def llistSum(l):
2    resSoFar = 0
3    while not isEmpty(l):
4      resSoFar = resSoFar + first(l)
5      l = rest(l)
6    return resSoFar
```

```
l : LL(7, 2, 3, 9)
resSoFar : 7
line to exec : 5
```

Line 4 is executed and our state is updated, with `resSoFar` having its value changed.

Tracing a `while` loop

```
1  def llistSum(l):
2    resSoFar = 0
3    while not isEmpty(l):
4      resSoFar = resSoFar + first(l)
5      l = rest(l)
6    return resSoFar
```

```
l : LL(2, 3, 9)
resSoFar : 7
line to exec : 3
```

Line 5 is executed and our state is updated again, this time changing `l`. The next line to execute is line 3 because at the end of the statements within a `while` statement we always continue execution back at the beginning of the loop.

Tracing a `while` loop

```
1  def llistSum(l):
2    resSoFar = 0
3    while not isEmpty(l):
4      resSoFar = resSoFar + first(l)
5      l = rest(l)
6    return resSoFar
```

```
l : LL(2, 3, 9)
resSoFar : 7
line to exec : 4
```

Line 3 is executed, and once again our condition is true so we continue with executing the statements of our loop.

Tracing a `while` loop

```
1  def llistSum(l):
2    resSoFar = 0
3    while not isEmpty(l):
4      resSoFar = resSoFar + first(l)
5      l = rest(l)
6    return resSoFar
```

```
l : LL(2, 3, 9)
resSoFar : 9
line to exec : 5
```

Line 4 is executed again updating the variable `resSoFar`.

Tracing a `while` loop

```
1  def llistSum(l):
2    resSoFar = 0
3    while not isEmpty(l):
4      resSoFar = resSoFar + first(l)
5      l = rest(l)
6    return resSoFar
```

```
l : LL(3, 9)
resSoFar : 9
line to exec : 3
```

Line 5 is executed again updating the variable `l`. Once again we continue back to the beginning of the loop.

Tracing a `while` loop

```
1  def llistSum(l):
2    resSoFar = 0
3    while not isEmpty(l):
4      resSoFar = resSoFar + first(l)
5      l = rest(l)
6    return resSoFar
```

```
l : LL(3, 9)
resSoFar : 9
line to exec : 4
```

Line 3 is executed again, and the loop condition is still true, so we continue onto line 4.

Tracing a `while` loop

```
1  def llistSum(l):
2    resSoFar = 0
3    while not isEmpty(l):
4      resSoFar = resSoFar + first(l)
5      l = rest(l)
6    return resSoFar
```

```
l : LL(3, 9)
resSoFar : 12
line to exec : 5
```

Line 4 is executed again, this time adding three to `resSoFar`.

Tracing a `while` loop

```
1  def llistSum(l):
2    resSoFar = 0
3    while not isEmpty(l):
4      resSoFar = resSoFar + first(l)
5      l = rest(l)
6    return resSoFar
```

```
l : LL(9)
resSoFar : 12
line to exec : 3
```

Line 5 is executed again updating `l`.

Tracing a `while` loop

```
1  def llistSum(l):
2    resSoFar = 0
3    while not isEmpty(l):
4      resSoFar = resSoFar + first(l)
5      l = rest(l)
6    return resSoFar
```

```
l : LL(9)
resSoFar : 12
line to exec : 4
```

Line 3 is executed again, and once again the expression is true, so we continue on into the loop.

Tracing a `while` loop

```
1  def llistSum(l):
2    resSoFar = 0
3    while not isEmpty(l):
4      resSoFar = resSoFar + first(l)
5      l = rest(l)
6    return resSoFar
```

```
l : LL(9)
resSoFar : 21
line to exec : 5
```

Line 4 is executed updating `resSoFar` by adding nine to it.

Tracing a `while` loop

```
1  def llistSum(l):
2    resSoFar = 0
3    while not isEmpty(l):
4      resSoFar = resSoFar + first(l)
5      l = rest(l)
6    return resSoFar
```

```
l : LL()
resSoFar : 21
line to exec : 3
```

Line 5 is executed updating `l` again, this time it is the empty `LList`! However, it is important to note that we still continue back up to line three, as that is where the check must happen to decide if we loop again.

Tracing a `while` loop

```
1  def llistSum(l):
2    resSoFar = 0
3    while not isEmpty(l):
4      resSoFar = resSoFar + first(l)
5      l = rest(l)
6    return resSoFar
```

```
l : LL()
resSoFar : 21
line to exec : 6
```

Line 3 is executed, this time the condition is false and so we do *not* enter the loop body. Instead execution continues on after the loop which is at line 6.

We have reached our return statement and can see based on our state that our function returns 21.

Solving problems iteratively

Practice Question: Recall the function we wrote `balancedGlyphs`, whose details are provided at the end of the searching slides.

Write the function `balancedGlyphs` without using recursion or higher-order functions. For any repetition you need to perform you should use while loops.

for statement

Python has another mechanism for iteration called the `for` loop. A `for` loop in Python takes the following form:

```
for identifer in expr:
  statement_1
  statement_2
  statement_n
```

`for` statement

Python has another mechanism for iteration called the `for` loop. A `for` loop in Python takes the following form:

```
for identifer in expr:
  statement_1
  statement_2
  statement_n
```

- `identifier` must be a valid Python identifier

`for` statement

Python has another mechanism for iteration called the `for` loop. A `for` loop in Python takes the following form:

```
for identifer in expr:
  statement_1
  statement_2
  statement_n
```

- `identifier` must be a valid Python identifier
- `expr` must be an expression that evaluates to a sequence

`for` statement

Python has another mechanism for iteration called the `for` loop. A `for` loop in Python takes the following form:

```
for identifer in expr:
  statement_1
  statement_2
  statement_n
```

- `identifier` must be a valid Python identifier
- `expr` must be an expression that evaluates to a sequence
- There must be at least one statement contained with the for loop, but could be arbitrarily more

Semantics of a Python `for` loop

```
for identifer in expr:
  LoopBody
```

The behaviour of a Python `for` loop is as follows

1. `expr` is evaluated once before the loop is executed, and results in a sequence of the form $(v_0, v_1, ..., v_n)$
2. If `identifier` has been bound to each item in the sequence then the for loop body is not executed, and code continues after the for loop and the subsequent steps do not occur.
3. `identifier` is bound to the *next* item in the sequence, the first time this is to v_0, each subsequent time this step is reached it will be to v_{i+1} where v_i is element it was bound to before reaching this step
4. The statements of the loop body execute, when execution reaches the end of the loop body execution continues back to step two.

`for` and `while`

Python's `for` loops are very handy when we know we want to perform an operation once for each item in a sequence, as the `for` loop implicitly handles the loop update for us that progresses us through the sequence.

On the other hand `while` loops are handy for when we want to loop not necessarily just over a sequence but until an arbitrary condition is met. With `while` loops the programmer must be careful to make sure their loop condition will eventually become false and terminate.

Writing `llistSum` with a `for` loop

We can rewrite `llistSum` to use a for loop instead of a while loop.

```
def llistSum(l):
  resSoFar = 0
  for num in l:
    resSoFar = resSoFar + num
  return resSoFar
```

Note: yes our `LLists` can be iterated over with a `for` loop!

Iteration and recursion

While it is true that iteration and recursion are computationally equivalent, that is everything that can be computed using recursion could be computed using iteration and vice versa, there are times when one type of solution is easier or more natural to write.

Determining when recursion would make solving a problem easier or when iteration would make solving a problem easier is a skill that takes time to develop. However, if the data you are operating on is recursively defined then recursion is a very natural choice as we've seen!

A hard problem

Practice Question: We now will try to solve a problem we just solved recursively iteratively instead. For this question you are to attempt to write the function `inBTreeToLList` that takes a single `BinaryTree` parameter and returns a `LList` that is the result of performing an in-order traversal of that tree, just as it was defined in the previous set of slides.

In your solution you may not use recursion or any higher-order functions. You must use only while or for loops for all repetition that is necessary.

Note: spend some time trying to solve this problem, but not *too* much. You may not be able to complete it, not because it is not possible but because this problem is much harder to solve iteratively than it is recursively!

The nature of recursion

A lot of the data that we work in computer science is often recursively defined. Trees are certainly a recursively defined data type as we've just studied. Recursion shows up all the time in how data is defined because recursion shows up in nature constantly.

When the data one is working on is recursively defined the natural solution to solving the problem is to tackle it recursively, since the problem can be easily broken up into subproblems based on the data definition.

In order to understand how to write an iterative solution to `inBTreeToLList` we must understand what advantages recursion is giving us, so that we can simulate them.

Table of Contents

Understanding recursion

So far we have understood recursion at the abstract level of functions. We've understood that if a function is defined to produce a certain value then we can recursively use it to get that result on a smaller subproblem.

Understanding recursion

So far we have understood recursion at the abstract level of functions. We've understood that if a function is defined to produce a certain value then we can recursively use it to get that result on a smaller subproblem.

But what actually happens when functions are called recursively? How does Python keep track of the data each individual function call holds, how does Python keep track of where to place the result of a function call?

Understanding recursion

So far we have understood recursion at the abstract level of functions. We've understood that if a function is defined to produce a certain value then we can recursively use it to get that result on a smaller subproblem.

But what actually happens when functions are called recursively? How does Python keep track of the data each individual function call holds, how does Python keep track of where to place the result of a function call?

We will now work with an abstract view of what a programming language must do to implement this behaviour, and remove some of the abstraction in cmput 275.

The problem — returning from functions

In our Python programs one statement is executed at a time — if a statement contains within it an expression that is a function call however then that function call must be evaluated to complete the execution of the statement.

So, when executing a statement if a function must be called then the Python interpreter must pause the statement, remember what context to return to, and execute the function to get a result.

It is also likely that in the process of executing that function another function call takes place, so that execution must be paused until the called function can return a final value. And the process may repeat.

The solution — the call stack

So, whenever a function call is performed the result must be returned to the statement that invoked the function, which means the context of the program at that point in time must be remembered. However, through the execution of that function further functions may be called and the context *they* return to must also be remembered.

We will present a very high-level abstract view of how the interpreter solves this problem. The interpreter solves this problem using the *call stack*.

What is the call stack?

The call stack is a data structure in which the Python interpreter will maintain the contexts of all the function calls currently awaiting a return value in the order they were called.

Each time a function is called the Python interpreter will add to the call stack what is called a *stack frame* for that function which stores the context of that running function. Whenever a function returns the stack frame for that function is removed from the call stack. In this way, the stack frame is an ordered collection of stack frames for unfinished function calls that are stored in the order in which they were called.

What is stored in a stack frame?

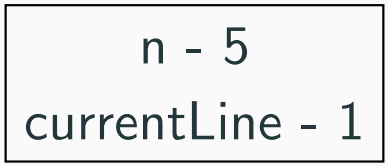
We will define a simplistic abstraction of what is stored in a stack frame for each function call.

In our abstraction each stack frame will store the value of each variable defined in that function call as well as the line of that function that is being executed.

Example call stack

Now we will consider a function call, how it gets executed, and how the call stack grows and shrinks during execution. In this case we will consider the execution of the function call `collatz(5)`

```
1  def collatz(n):
2    if n == 1:
3      return 1
4    if n%2 == 1:
5      ror = collatz(3*n + 1)
6    else:
7      ror = collatz(n//2)
8    return 1 + ror
```



Example call stack

As the function call `collatz(5)` executes it eventually reaches the line 5 which is a assignment statement, however the expression on the right hand side of the assignment includes a function call which must be executed. As such, this function call is paused until it receives its answer from the newly called function which must be pushed onto the call stack.

```
1  def collatz(n):
2    if n == 1:
3      return 1
4    if n%2 == 1:
5      ror = collatz(3*n + 1)
6    else:
7      ror = collatz(n//2)
8    return 1 + ror
```

n = 16 currentLine = 1
n = 5 currentLine = 5

Example call stack

In the execution of `collatz(16)` however we reach line 7, which calls `collatz(8)`. So we must pause this executing function call until the newly called function returns with an answer.

```
1  def collatz(n):
2    if n == 1:
3      return 1
4    if n%2 == 1:
5      ror = collatz(3*n + 1)
6    else:
7      ror = collatz(n//2)
8    return 1 + ror
```

n = 8 currentLine = 1
n = 16 currentLine = 7
n = 5 currentLine = 5

Example call stack

Again, in executing `collatz(8)` we reach line 7 which must resolve the function call `collatz(4)`.

```
1  def collatz(n):
2    if n == 1:
3      return 1
4    if n%2 == 1:
5      ror = collatz(3*n + 1)
6    else:
7      ror = collatz(n//2)
8    return 1 + ror
```

n = 4 currentLine = 1
n = 8 currentLine = 7
n = 16 currentLine = 7
n = 5 currentLine = 5

Example call stack

Again, in executing `collatz(4)` we reach line 7 which must resolve the function call `collatz(2)`.

```
1  def collatz(n):
2    if n == 1:
3      return 1
4    if n%2 == 1:
5      ror = collatz(3*n + 1)
6    else:
7      ror = collatz(n//2)
8    return 1 + ror
```

n = 2 currentLine = 1
n = 4 currentLine = 7
n = 8 currentLine = 7
n = 16 currentLine = 7
n = 5 currentLine = 5

Example call stack

Again, in executing `collatz(2)` we reach line 7 which must resolve the function call `collatz(1)`.

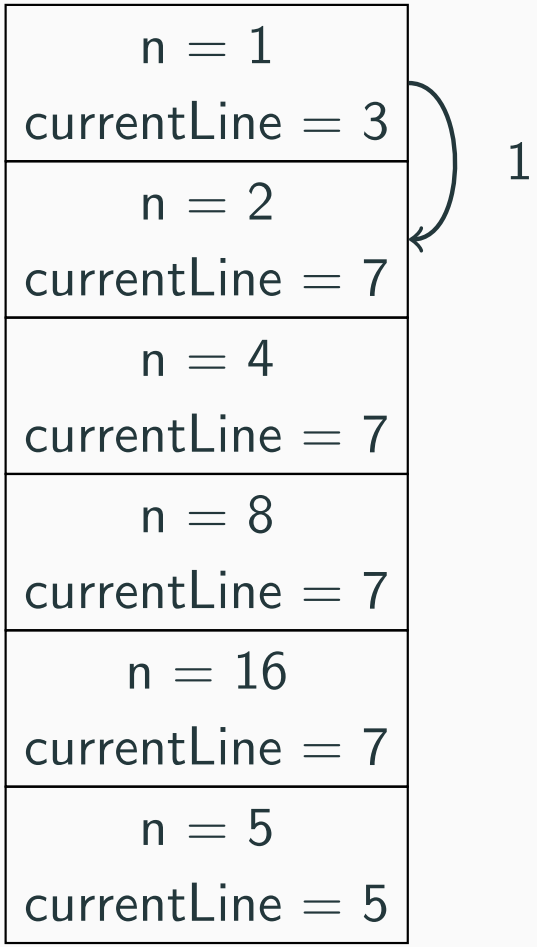
```
1  def collatz(n):
2    if n == 1:
3      return 1
4    if n%2 == 1:
5      ror = collatz(3*n + 1)
6    else:
7      ror = collatz(n//2)
8    return 1 + ror
```

n = 1 currentLine = 1
n = 2 currentLine = 7
n = 4 currentLine = 7
n = 8 currentLine = 7
n = 16 currentLine = 7
n = 5 currentLine = 5

Example call stack

Now in the execution of `collatz(1)` we reach line 3 and have to return the answer to the function that called us, luckily that is the stack frame directly below us on the call stack.

```
1  def collatz(n):
2    if n == 1:
3      return 1
4    if n%2 == 1:
5      ror = collatz(3*n + 1)
6    else:
7      ror = collatz(n//2)
8    return 1 + ror
```



Example call stack

Since `collatz(1)` returned, its execution is finished and its frame is removed from the call stack. The interpreter knows where to return the resultant value to because of the context on the previous stack frame, and so `ror` is assigned to the return value of the function call.

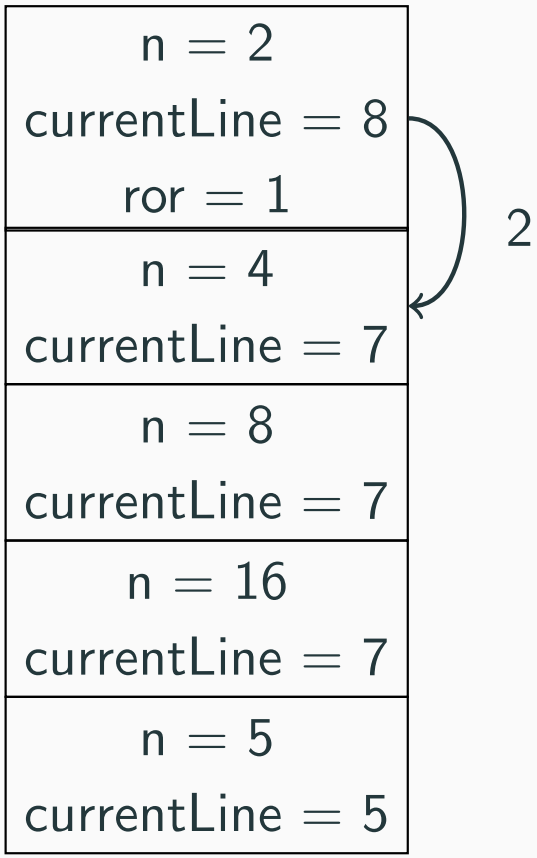
```
1  def collatz(n):
2    if n == 1:
3      return 1
4    if n%2 == 1:
5      ror = collatz(3*n + 1)
6    else:
7      ror = collatz(n//2)
8    return 1 + ror
```

n = 2 currentLine = 7 ror = 1
n = 4 currentLine = 7
n = 8 currentLine = 7
n = 16 currentLine = 7
n = 5 currentLine = 5

Example call stack

So `collatz(2)` resumes execution and reaches line 8 where it finally returns a value.

```
1  def collatz(n):
2    if n == 1:
3      return 1
4    if n%2 == 1:
5      ror = collatz(3*n + 1)
6    else:
7      ror = collatz(n//2)
8    return 1 + ror
```



Example call stack

So `collatz(2)` has finished and as such its stack frame is removed and execution of `collatz(4)` can continue after receiving the result of the function it called.

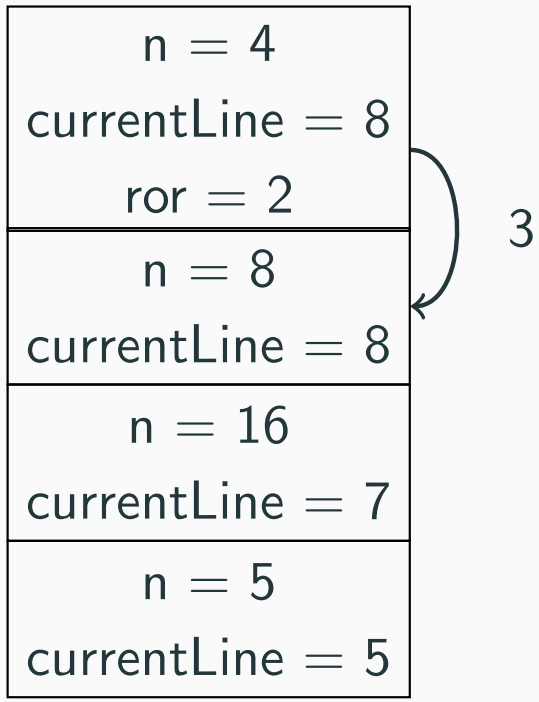
```
1  def collatz(n):
2    if n == 1:
3      return 1
4    if n%2 == 1:
5      ror = collatz(3*n + 1)
6    else:
7      ror = collatz(n//2)
8    return 1 + ror
```

n = 4 currentLine = 7 ror = 2
n = 8 currentLine = 7
n = 16 currentLine = 7
n = 5 currentLine = 5

Example call stack

`collatz(4)` resumes execution and reaches line 8 where it finally returns a value.

```
1  def collatz(n):
2    if n == 1:
3      return 1
4    if n%2 == 1:
5      ror = collatz(3*n + 1)
6    else:
7      ror = collatz(n//2)
8    return 1 + ror
```



Example call stack

With `collatz(4)` complete the function call `collatz(8)` finishes evaluating the function call and can continue execution.

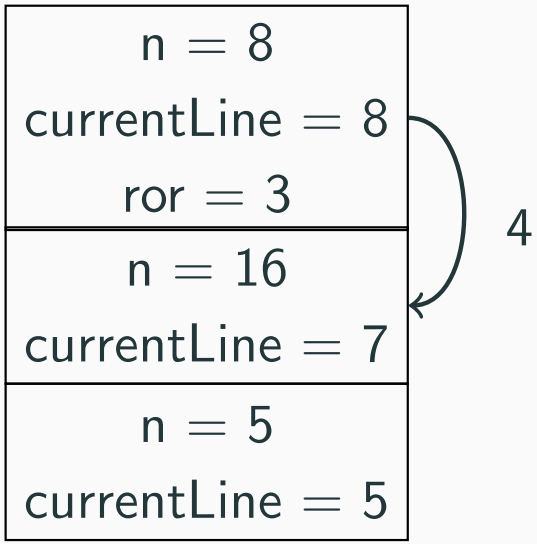
```
1  def collatz(n):
2    if n == 1:
3      return 1
4    if n%2 == 1:
5      ror = collatz(3*n + 1)
6    else:
7      ror = collatz(n//2)
8    return 1 + ror
```

n = 8 currentLine = 7 ror = 3
n = 16 currentLine = 7
n = 5 currentLine = 5

Example call stack

`collatz(8)` reaches line 8 and finshes execution returning the value four.

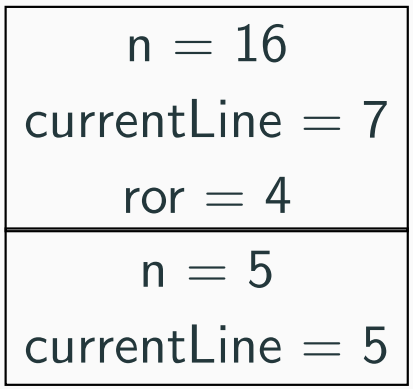
```
1  def collatz(n):
2    if n == 1:
3      return 1
4    if n%2 == 1:
5      ror = collatz(3*n + 1)
6    else:
7      ror = collatz(n//2)
8    return 1 + ror
```



Example call stack

With `collatz(8)` complete it has its stack frame removed from the stack when its value is returned to `collatz(16)` which can resume execution.

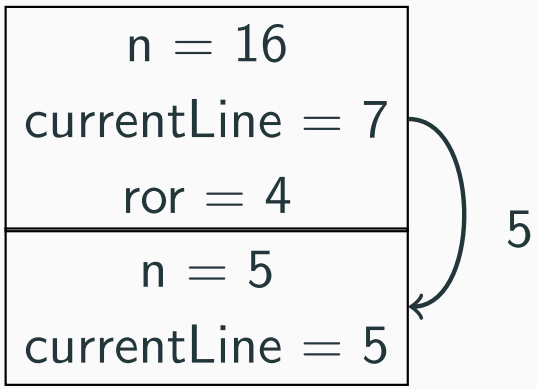
```
1  def collatz(n):
2    if n == 1:
3      return 1
4    if n%2 == 1:
5      ror = collatz(3*n + 1)
6    else:
7      ror = collatz(n//2)
8    return 1 + ror
```



Example call stack

`collatz(16)` reaches line 8 and also returns its final value of five to the function that called it.

```
1  def collatz(n):
2    if n == 1:
3      return 1
4    if n%2 == 1:
5      ror = collatz(3*n + 1)
6    else:
7      ror = collatz(n//2)
8    return 1 + ror
```



Example call stack

`collatz(5)` *finally* receives the value for its function call to `collatz(16)` and can resume execution.

```
1  def collatz(n):
2    if n == 1:
3      return 1
4    if n%2 == 1:
5      ror = collatz(3*n + 1)
6    else:
7      ror = collatz(n//2)
8    return 1 + ror
```

n = 5
currentLine = 5
ror=5

Example call stack

In its execution `collatz(5)` reaches line 8 and can finally return its final value of six. In our call stack we have not included who called `collatz(5)`, though there will always be *someting* there to return to.

```
1  def collatz(n):
2    if n == 1:
3      return 1
4    if n%2 == 1:
5      ror = collatz(3*n + 1)
6    else:
7      ror = collatz(n//2)
8    return 1 + ror
```

n = 5
currentLine = 8
ror=5

Recursion and in-order traversal

So, in our recursive implementation of in-order traversal of a tree the call stack was keeping track of the state of each of our functions.

Recursion and in-order traversal

So, in our recursive implementation of in-order traversal of a tree the call stack was keeping track of the state of each of our functions.

This not only meant that we were able to recursively call our function on our subtrees to break the problem up into smaller sub problems, but it also meant that the call stack was remembering the state of where we were at in each function call so that we could return to the correct place!

Recursion and in-order traversal

So, in our recursive implementation of in-order traversal of a tree the call stack was keeping track of the state of each of our functions.

This not only meant that we were able to recursively call our function on our subtrees to break the problem up into smaller sub problems, but it also meant that the call stack was remembering the state of where we were at in each function call so that we could return to the correct place!

Or more simply put the call stack was remembering for us the context if we had already searched our left tree or not — since if we had already processed our left tree we should continue on to process our current node and then to process our right tree, however if the recursion we are returning from was on our right tree then we should continue on to simply combine our answer and return!

Solving iterative in-order traversal

So, in order to solve in-order traversal iteratively we must simulate what the call stack does implicitly for us which is remembering our current context, and the work that remains to be done.

To simulate what the call stack is doing for us, we must understand the data type that implements the call stack, which is the *Stack* ADT.

Table of Contents

Stack — an abstract data type

We recall that an Abstract Data Type (ADT) is a data type that is defined by the behaviour it provides and not by its implementation. As such, to define the stack ADT we need only to define the behaviour it provides which could be implemented any number of ways.

Abstractly the stack data type represents exactly that — a stacked collection of data.

Stack — analogy

Envision an empty table, this is our empty stack. Now, we can add items to our stack only by placing them on top of the stack.

Stack — analogy

Envision an empty table, this is our empty stack. Now, we can add items to our stack only by placing them on top of the stack.

If we place a book on the table then the book is our first, and only, item in our stack.

Stack — analogy

Envision an empty table, this is our empty stack. Now, we can add items to our stack only by placing them on top of the stack.

If we place a book on the table then the book is our first, and only, item in our stack.

Next, we may want to add a folded t-shirt to our stack. To do so we simply place the folded t-shirt on top of the book. Now, the folded t-shirt is at the *top* of our stack and the folded t-shirt is at the *bottom* of our stack.

Stack — analogy

Envision an empty table, this is our empty stack. Now, we can add items to our stack only by placing them on top of the stack.

If we place a book on the table then the book is our first, and only, item in our stack.

Next, we may want to add a folded t-shirt to our stack. To do so we simply place the folded t-shirt on top of the book. Now, the folded t-shirt is at the *top* of our stack and the folded t-shirt is at the *bottom* of our stack.

Now, if we wanted to access the book we cannot do so directly, we would have to first lift the t-shirt that sits on top of it — but to lift the t-shirt would be to take it off of the stack! Such is the nature of the stack ADT.

Stack — analogy

So, in our example we cannot access the book without first taking the t-shirt off of our stack. This is how a stack ADT works. In fact, we can only ever access the *top* item of a stack. Also, given how we add items to our stack the top item of the stack is always the most recently added item!

Stack — analogy

So, in our example we cannot access the book without first taking the t-shirt off of our stack. This is how a stack ADT works. In fact, we can only ever access the *top* item of a stack. Also, given how we add items to our stack the top item of the stack is always the most recently added item!

Due to this behaviour a stack is called a *last-in first-out (LIFO)*[1] data structure. It is called such because the **last** item to be added **in** to the stack is the **first** item to come **out** of the stack.

[1] Or, equivalently, *first-in last-out (FILO)*

Stack — analogy

So, in our example we cannot access the book without first taking the t-shirt off of our stack. This is how a stack ADT works. In fact, we can only ever access the *top* item of a stack. Also, given how we add items to our stack the top item of the stack is always the most recently added item!

Due to this behaviour a stack is called a *last-in first-out (LIFO)*[1] data structure. It is called such because the **last** item to be added **in** to the stack is the **first** item to come **out** of the stack.

So a stack can be envisioned as a pile of things where you can only add an item to the top of the pile, or remove an item from the top of the pile. To get to items further down in the pile you must repeatedly remove items from the top of the pile.

[1] Or, equivalently, *first-in last-out (FILO)*

Operations on a `Stack`

We will now define the functions we will use to define the behaviour of our `Stack` ADT.

Operations on a `Stack`

We will now define the functions we will use to define the behaviour of our `Stack` ADT.

- `emptyStack` — A constant function with zero parameters that returns an empty `Stack`

Operations on a `Stack`

We will now define the functions we will use to define the behaviour of our `Stack` ADT.

- `emptyStack` — A constant function with zero parameters that returns an empty `Stack`
- `push` — A function that takes a `Stack` parameter and a value and returns a new `Stack` that is the result of adding the given value to the top of the given `Stack`

Operations on a `Stack`

We will now define the functions we will use to define the behaviour of our `Stack` ADT.

- `emptyStack` — A constant function with zero parameters that returns an empty `Stack`
- `push` — A function that takes a `Stack` parameter and a value and returns a new `Stack` that is the result of adding the given value to the top of the given `Stack`
- `peek` — A function that takes a single `Stack` parameter and returns the value that is stored at the top of the stack

Operations on a `Stack`

We will now define the functions we will use to define the behaviour of our `Stack` ADT.

- `emptyStack` — A constant function with zero parameters that returns an empty `Stack`
- `push` — A function that takes a `Stack` parameter and a value and returns a new `Stack` that is the result of adding the given value to the top of the given `Stack`
- `peek` — A function that takes a single `Stack` parameter and returns the value that is stored at the top of the stack
- `pop` — A function that takes a single `Stack` parameter and returns a new `Stack` that is the result of removing the top item of the given `Stack`

Operations on a `Stack`

We will now define the functions we will use to define the behaviour of our `Stack` ADT.

- `emptyStack` — A constant function with zero parameters that returns an empty `Stack`
- `push` — A function that takes a `Stack` parameter and a value and returns a new `Stack` that is the result of adding the given value to the top of the given `Stack`
- `peek` — A function that takes a single `Stack` parameter and returns the value that is stored at the top of the stack
- `pop` — A function that takes a single `Stack` parameter and returns a new `Stack` that is the result of removing the top item of the given `Stack`
- `stackIsEmpty` — A function that takes a single `Stack` parameter and returns True if that `Stack` is empty and False otherwise.

Examples usage of `Stack` functions

We will now provide example applications of our `Stack` functions. In order to do so we must be able to visually represent stacks, and will do so as a sequence of data contained within two less than characters. We choose less than characters because they will always point to the top of the stack, so the top of the stack is the left-most value shown.

For example the stack with a lamp on top of a t-shirt that is on top of a book we would represent as:

```
<"lamp", "t-shirt", "book"<
```

Note that `<"lamp", "t-shirt", "book"<` is not an actual value we can write in Python but is simply how we are choosing to represent a value visually for communication.

Examples usage of `Stack` functions

```
emptyStack() -> <<
```

Examples usage of `Stack` functions

```
emptyStack() -> <<
push(5, emptyStack()) -> <5<
```

Examples usage of `Stack` functions

```
emptyStack() -> <<
push(5, emptyStack()) -> <5<
push(7, push(5, emptyStack())) -> <7, 5<
push(10, <3, 2, -5, 8<) -> <10, 3, 2, -5, 8<
```

Examples usage of `Stack` functions

```
emptyStack() -> <<
push(5, emptyStack()) -> <5<
push(7, push(5, emptyStack())) -> <7, 5<
push(10, <3, 2, -5, 8<) -> <10, 3, 2, -5, 8<
peek(<10, 3, 2, -5, 8<) -> 10
```

Examples usage of `Stack` functions

```
emptyStack() -> <<
push(5, emptyStack()) -> <5<
push(7, push(5, emptyStack())) -> <7, 5<
push(10, <3, 2, -5, 8<) -> <10, 3, 2, -5, 8<
peek(<10, 3, 2, -5, 8<) -> 10
pop(<"a", "b", "c"<) -> <"b", "c"<
```

Examples usage of `Stack` functions

```
emptyStack() -> <<
push(5, emptyStack()) -> <5<
push(7, push(5, emptyStack())) -> <7, 5<
push(10, <3, 2, -5, 8<) -> <10, 3, 2, -5, 8<
peek(<10, 3, 2, -5, 8<) -> 10
pop(<"a", "b", "c"<) -> <"b", "c"<
stackIsEmpty(<1, 2<) -> False
stackIsEmpty(<<) -> True
```

Implementing a `Stack`

So, a `Stack` is a collection data type from which we can start with an empty collection and build a collection only by adding items to the top of the collection.

Given a `Stack` we can only look at the top item of it or remove the top item of it to produce a `Stack` that is the same except without that top item.

Lastly, we can check if a `Stack` is empty or not.

Hmm, seems familiar.

Implementing a `Stack`

So, a ~~`Stack`~~ `LList` is a collection data type from which we can start with an empty collection and build a collection only by adding items to the ~~top~~ front of the collection.

Given a `Stack` we can only look at the top item of it or remove the top item of it to produce a `Stack` that is the same except without that top item.

Lastly, we can check if a `Stack` is empty or not.

Hmm, seems familiar.

Implementing a `Stack`

So, a ~~`Stack`~~ `LList` is a collection data type from which we can start with an empty collection and build a collection only by adding items to the ~~top~~ front of the collection.

Given a ~~`Stack`~~ `LList` we can only look at the ~~top~~ first item of it or remove the ~~top~~ first item of it to produce a ~~`Stack`~~ `LList` that is the same except without that ~~top~~ first item.

Lastly, we can check if a `Stack` is empty or not.

Hmm, seems familiar.

Implementing a `Stack`

So, a ~~`Stack`~~ `LList` is a collection data type from which we can start with an empty collection and build a collection only by adding items to the ~~top~~ front of the collection.

Given a ~~`Stack`~~ `LList` we can only look at the ~~top~~ first item of it or remove the ~~top~~ first item of it to produce a ~~`Stack`~~ `LList` that is the same except without that ~~top~~ first item.

Lastly, we can check if a ~~`Stack`~~ `LList` is empty or not.

So we can use a `LList` to represent a `Stack`!

Example `Stack` implementation

In order to make it clear in our examples that we are in fact using a `Stack` we want to define the data type and the functions on it. This can be done simply as shown below:

```
'''
A Stack of X is one of
  - emptyStack()
  - push(X, Stack of X)
'''
emptyStack = empty
push = cons
pop = rest
peek = first
stackIsEmpty = isEmpty
```

Implementing in-order traversal

Now that we have our `Stack` ADT and understand how to use it we can try to solve in-order traversal iteratively.

```
def inBTreeToLList(t):
```

Implementing in-order traversal

We must start with our loop initialization. Since we are simulating a stack, the data we must initialize is a stack. Much like our call stack the top item of the stack will should represent the work we have remaining to do. If ever the stack is empty then that indicates there is no more work to perform!

We don't want an empty stack to begin with, because that implies the work is done. So we should place our tree `t` into the stack as the first item to process.

```
def inBTreeToLList(t):
  cStack = push(t, emptyStack())
```

Implementing in-order traversal

We are also planning to build a `LList` out of our tree, and as such should initialize that `LList`.

Since we have not processed any nodes yet, that `LList` should begin empty.

```
def inBTreeToLList(t):
  cStack = push(t, emptyStack())
  result = empty()
```

Implementing in-order traversal

Just as our call stack grew as we recursively called ourselves our simulated stack will grow as we push subproblems to solve onto it.

Our simulated stack will shrink whenever we complete work and pop it off. As such, our loop termination condition will be when our stack is empty.

```
def inBTreeToLList(t):
  cStack = push(t, emptyStack())
  result = empty()
  while not stackIsEmpty(cStack):
```

Implementing in-order traversal

Now, our loop body effectively represents each recursive application of the function.

The first item of our stack is the tree to process which is effectively the argument provided to our recursion, so we should pull out the first item of the stack and consider how to process it as we would in a recursive function.

```
def inBTreeToLList(t):
  cStack = push(t, emptyStack())
  result = empty()
  while not stackIsEmpty(cStack):
```

Implementing in-order traversal

So we pull out our tree to process by peeking at the top of the stack, and then updating the stack by popping the item we just pulled out.

```
def inBTreeToLList(t):
  cStack = push(t, emptyStack())
  result = empty()
  while not stackIsEmpty(cStack):
    tree = peek(cStack)
    cStack = pop(cStack)
```

Implementing in-order traversal

Now we have a tree to process in-order traversal of regularly, so first we should check if we have a left child as the left subtree is processed first in an in-order traversal.

```
def inBTreeToLList(t):
  cStack = push(t, emptyStack())
  result = empty()
  while not stackIsEmpty(cStack):
    tree = peek(cStack)
    cStack = pop(cStack)
    if tree[1] != ():
```

Implementing in-order traversal

If our left subtree is not empty then we must "recursively call ourselves". In order to simulate this we must modify our stack in the way the call stack would be modified.

To simulate recursion we can push our recursive argument onto the stack, however we also must remember the context to return to, so before we push our recursive argument we must push the contex that represents the work still to be done on this node.

```
def inBTreeToLList(t):
  cStack = push(t, emptyStack())
  result = empty()
  while not stackIsEmpty(cStack):
    tree = peek(cStack)
    cStack = pop(cStack)
    if tree[1] != ():
```

Implementing in-order traversal

Well, since we are going to be pushing our left subtree as the next work to be done that means the work to "return to" after that will be everything in this current tree after processing the left subtree.

So, we can store the context of this "call" as a modified version of the current tree without its left subtree.

```
def inBTreeToLList(t):
  cStack = push(t, emptyStack())
  result = empty()
  while not stackIsEmpty(cStack):
    tree = peek(cStack)
    cStack = pop(cStack)
    if tree[1] != ():
```

Implementing in-order traversal

Once we have pushed our context and the next "call" onto our stack, we want to begin what would be the next recursive call which would be the beginning of our loop. So, the rest of the code in our loop should be contained with an else — which is the code that runs only when we *don't* have left child.

```
def inBTreeToLList(t):
  cStack = push(t, emptyStack())
  result = empty()
  while not stackIsEmpty(cStack):
    tree = peek(cStack)
    cStack = pop(cStack)
    if tree[1] != ():
      cxt = (tree[0], (), tree[2])
      cStack = push(cxt, cStack)
      cStack = push(tree[1], cStack)
```

Implementing in-order traversal

Now, in our `else` clause we must consider what to do if the work to process is a `node` that has no left child.

In that case, we should simply process the current node and then "recursively" process the right subtree.

```
def inBTreeToLList(t):
  cStack = push(t, emptyStack())
  result = empty()
  while not stackIsEmpty(cStack):
    tree = peek(cStack)
    cStack = pop(cStack)
    if tree[1] != ():
      cxt = (tree[0], (), tree[2])
      cStack = push(cxt, cStack)
      cStack = push(tree[1], cStack)
    else:
```

Implementing in-order traversal

Processing the current node is simply taking the value we have and appending it to the end of our result. We will assume we have the function append which takes an item and a `LList` and returns a new `LList` that is the result of appending that item to the back of the given `LList`.

```
def inBTreeToLList(t):
  cStack = push(t, emptyStack())
  result = empty()
  while not stackIsEmpty(cStack):
    tree = peek(cStack)
    cStack = pop(cStack)
    if tree[1] != ():
      cxt = (tree[0], (), tree[2])
      cStack = push(cxt, cStack)
      cStack = push(tree[1], cStack)
    else:
      result = append(tree[0], result)
```

Implementing in-order traversal

After processing our current node we must "recursively" process our right subtree. That is, we must push our right subtree onto our stack for processing.

```
def inBTreeToLList(t):
  cStack = push(t, emptyStack())
  result = empty()
  while not stackIsEmpty(cStack):
    tree = peek(cStack)
    cStack = pop(cStack)
    if tree[1] != ():
      cxt = (tree[0], (), tree[2])
      cStack = push(cxt, cStack)
      cStack = push(tree[1], cStack)
    else:
      result = append(tree[0], result)
      cStack = push(tree[2], cStack)
```

Implementing in-order traversal

However, if our right subtree is empty then we pushed an empty tree onto our stack and our loop does not currently properly handle that. We can solve this by simply not pushing our right subtree if it is indeed empty.

```
def inBTreeToLList(t):
  cStack = push(t, emptyStack())
  result = empty()
  while not stackIsEmpty(cStack):
    tree = peek(cStack)
    cStack = pop(cStack)
    if tree[1] != ():
      cxt = (tree[0], (), tree[2])
      cStack = push(cxt, cStack)
      cStack = push(tree[1], cStack)
    else:
      result = append(tree[0], result)
      cStack = push(tree[2], cStack)
```

Implementing in-order traversal

Lastly, we must return our value once our loop is complete.

```
def inBTreeToLList(t):
  cStack = push(t, emptyStack())
  result = empty()
  while not stackIsEmpty(cStack):
    tree = peek(cStack)
    cStack = pop(cStack)
    if tree[1] != ():
      cxt = (tree[0], (), tree[2])
      cStack = push(cxt, cStack)
      cStack = push(tree[1], cStack)
    else:
      result = append(tree[0], result)
      if tree[2] != ():
        cStack = push(tree[2], cStack)
  return result
```

Strengths of recursion and iteration

This iterative solution to in-order traversal of a tree is much more complicated than the recursive solution.

This problem illustrates the fact that some problems are more easily solved with recursion.

Similarly, some problems are more easily solved with iteration.

Strengths of recursion and iteration

As a general rule of thumb, though there are exceptions and personal preference:

- If working with a recursive data type then a recursive solution may be easier
- Iteration may be easier when needing to walk the items of an ordered collection in the order they appear in that collection

Iteration over indices

Practice Question: write the function `maximalIndex` that takes a tuple of integers and returns the index of the largest element in that tuple. For example:

```
maximalIndex((1, 20, 3, -5)) -> 1
maximalIndex((-32, 500, 323, 1043, 108)) -> 3
```

However, you must solve the problem using only `for` loops for repetition, no `while` loops, recursion, or higher-order functions except for `buildList` which you not only may use but will need to use.

Solving pre-order and post-order traversal

Practice Question: write the functions `preBTreeToLList` and `postBTreeToLList` as they were defined in the previous set of slides but using only iteration and no recursion.