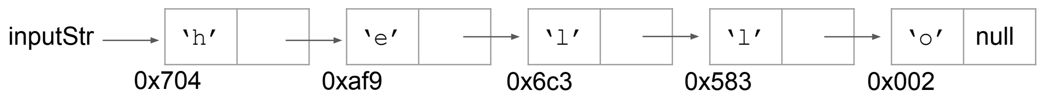
Section 5

CSCI E-22

Will Begin Shortly

Tracing through `StringNode.print()`

Let’s take a look at the execution of the `StringNode.print()` method where the input list looks like this (the hexadecimal number next to each node is the memory location of that node):



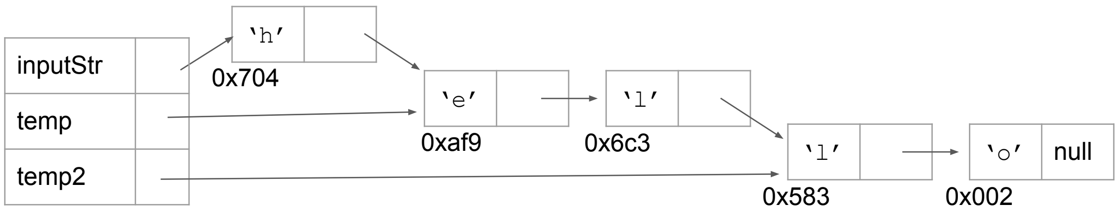
We want to draw a diagram which shows how the stack evolves with each subsequent call, like we did on the written portion of Problem Set 1. The important things to track are the output so far, and the address the current `str` variable holds in each frame.

Tracing through `StringNode.print()`

```java
/*
 * Recursively process each node of
 * a linked list formed by
 * StringNodes, printing each one.
 */
public static void print(StringNode str)
{
    if (str == null) {
        return;
    } else {
        System.out.print(str.ch);
        print(str.next);
    }
}
```

Output:

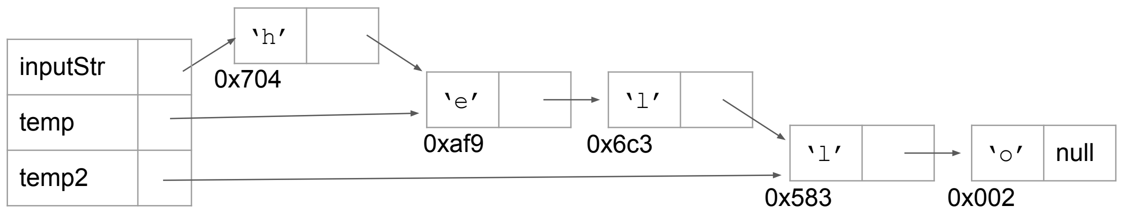
More practice with variable expressions and references



Specify how the following expressions evaluate:

- What is the value of `temp.next`?
- What does `temp.next.ch` evaluate to?
- What does `inputStr.next.next == temp` evaluate to?
- What are some ways we can access the character `'o'`?

More practice with variable expressions and references



How do the following statements change the diagram?

- temp.next = temp2
- temp = temp2.next.next
- inputStr = inputStr.next
- temp2 = null

Converting a simple recursive method to an iterative one

```java
/**
 * numOccurrences - find the number of
 * occurrences of the character ch in
 * the linked list to which str refers
 */
public static int numOccurrences(StringNode str,  char ch) {
    if (str == null)
        return 0;

    int occurInRest = numOccurrences(str.next, ch);

    if (str.ch == ch)
        return 1 + occurInRest;
    else
        return occurInRest;
}
```

This is the recursive version of the `numOccurrences()` method which finds the number of occurrences of a specified character in our linked list.

Converting a simple recursive method to an iterative one

```
public static int numOccurrences(StringNode str,  char ch) {
    if (str == null)
        return 0;
    int occurInRest = numOccurrences(str.next, ch);
    if (str.ch == ch)
        return 1 + occurInRest;
    else
        return occurInRest;
}
```

We want to convert this method into an iterative one. Think back to our trace through the `StringNode.print()` method. Do you see any similarities? What do you think the stack looks like when this method is executed? What does it return?

Converting the `StringNode.read()` method

The read method takes an `InputStream` object, which can represent any input source which is processed as a sequence of bytes (such as the `System.in` input stream which allows us to read from the keyboard, or a file or network resource). The `IOException` isn't important; it's simply there in case the input stream goes down when the method is executed. `StringNode.read(System.in)` will read any string the user enters one character at a time and return a reference to a linked list which it constructs as the user types each character. Here is the original, recursive `read()` method:

```
/**
 * read - reads a string from an input stream and returns a
 * reference to a linked list containing the characters in the string
 */
public static StringNode read(InputStream in) throws IOException {
    StringNode str;
    char ch = (char)in.read();

    if (ch == '\n')     // base case
        str = null;
    else
        str = new StringNode(ch, read(in));

    return str;
 }
```

Converting the `StringNode.read()` method

In what order does the `read()` method construct the linked list? Will this work as well in our iterative implementation? Why or why not?

Converting the `StringNode.read()` method

```
public static StringNode read(InputStream in) throws IOException {
    StringNode str;
    char ch = (char)in.read();

    if (ch == '\n')     // base case
        str = null;
    else
        str = new StringNode(ch, read(in));

    return str;
 }
```

Clearly we'll need a different strategy in the iterative implementation, since we'll need to start at the front of the list if we want to keep the input in the order in which it is read from the input stream. How many pointers will we need if we want the iterative method to add new elements to the back and then return the reference to the first node of the list?