3.1.1

Assume that the stack `names` is defined as in Figure 3.2(c) and perform the following sequence of operations. Indicate the result of each operation and show the new stack if it is changed.

```
names.push("Jane");
names.push("Joseph");
String top = names.pop();
String nextTop = names.peek();
```

Philip
Dustin
Robin
Debbie
Rich

```
names.push("Jane");
```

Jane
Philip
Dustin
Robin
Debbie
Rich

```
names.push("Joseph");
```

Joseph
Jane
Philip
Dustin
Robin
Debbie
Rich

```
String top = names.pop();
```

Jane
Philip
Dustin
Robin
Debbie
Rich

The String top contains "Joseph"

```
String nextTop = names.peek();
```

The String nextTop contains "Jane" and the stack remains unchanged.

3.1.3

What would be the effect of using `peek` instead of `pop` in Question 2?

The program would be in an infinite loop printing "`Philip`".

3.2.1

The result returned by the palindrome finder depends on all characters in a string, including spaces and punctuation. Discuss how you would modify the palindrome finder so that only the letters in the input string were used to determine whether the input string was a palindrome. You should ignore any other characters.

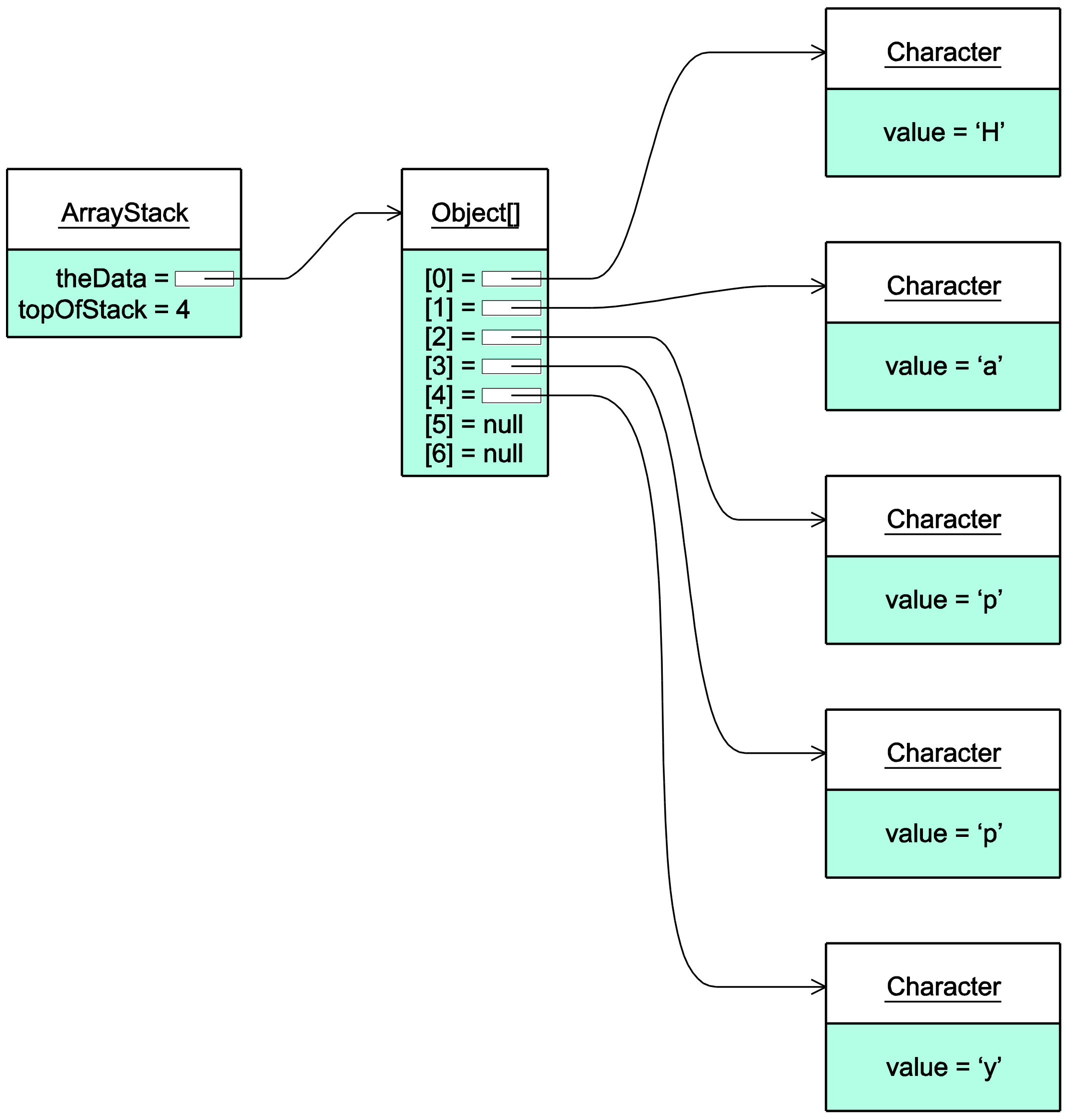
The palindrome finder could be modified to only push letters onto the stack and to append only the letter characters to a new temporary string being formed. Then, after the reverse string is built by emptying the stack, it could be compared to the temporary string.

3.3.1

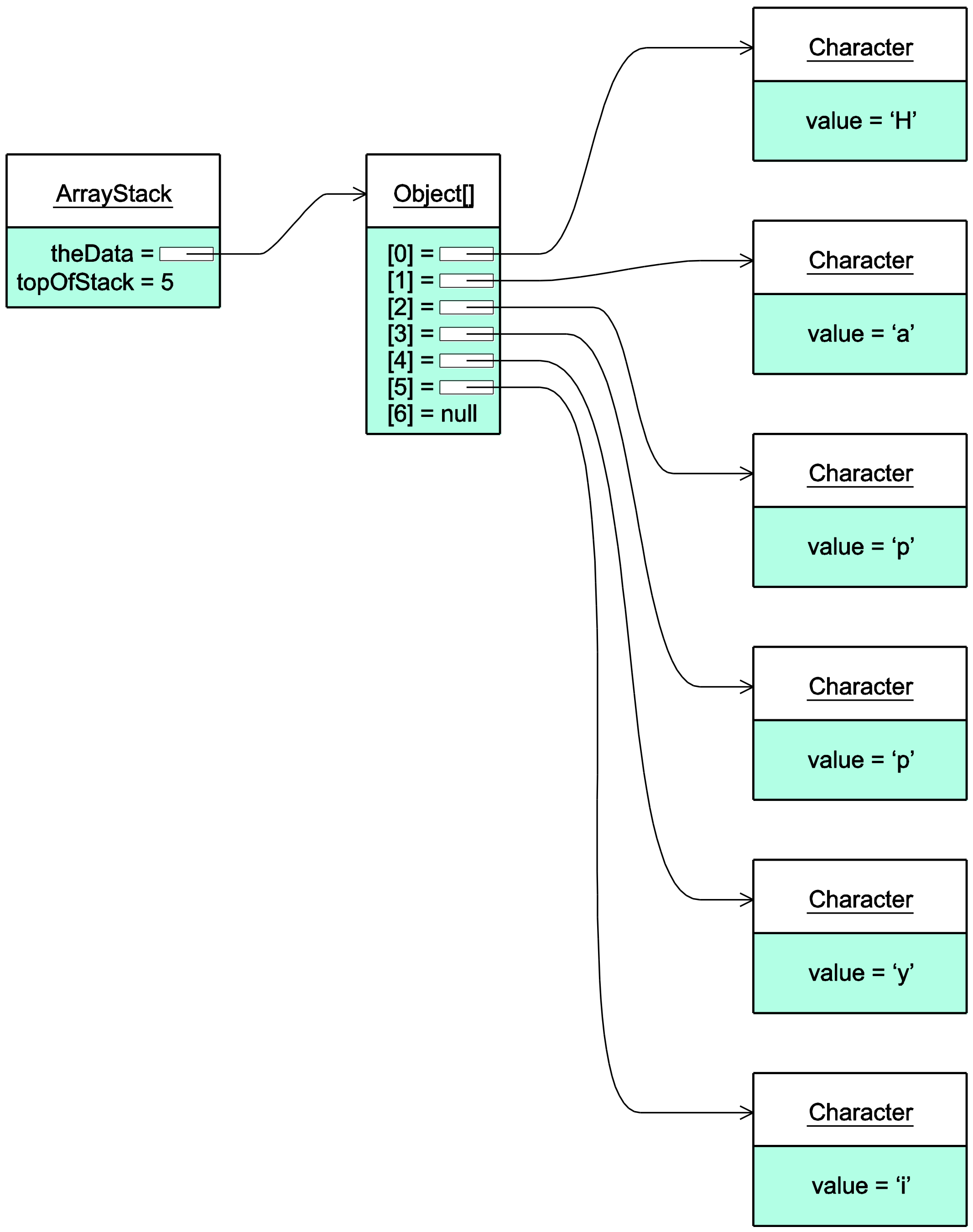
For the implementation of stack `s` using an array as the underlying data structure (see Figure 3.4), show how the underlying data structure changes after each statement below executes. What is the value of `topOfStack`? Assume the initial capacity of the stack is 7 and the characters in `"Happy"` are stored on the stack (`H` pushed on first).

```
s.push('i');
s.push('s');
char ch1 = s.pop();
s.pop();
s.push(' ');
char ch2 = s.peek();
```

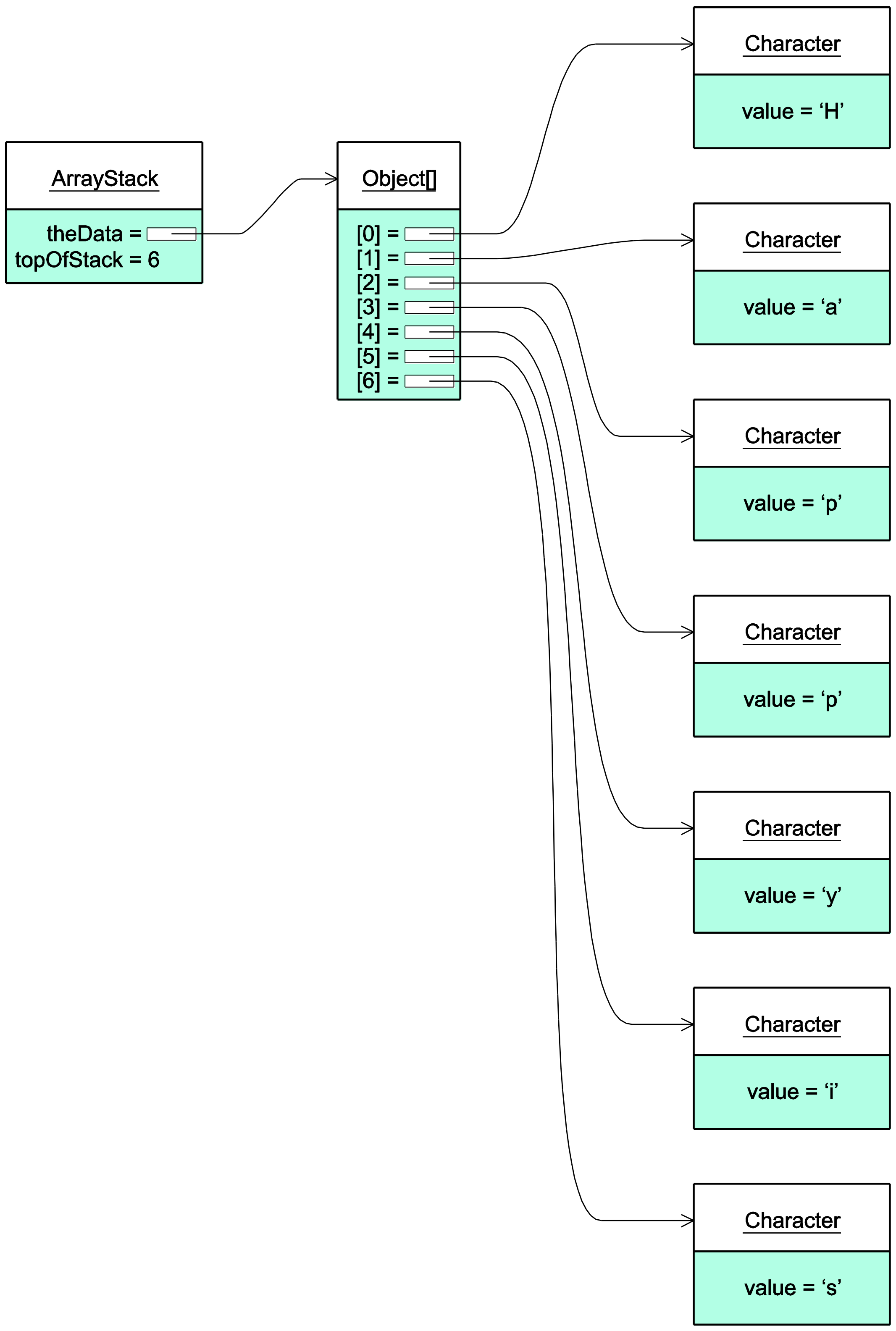
Initial stack:



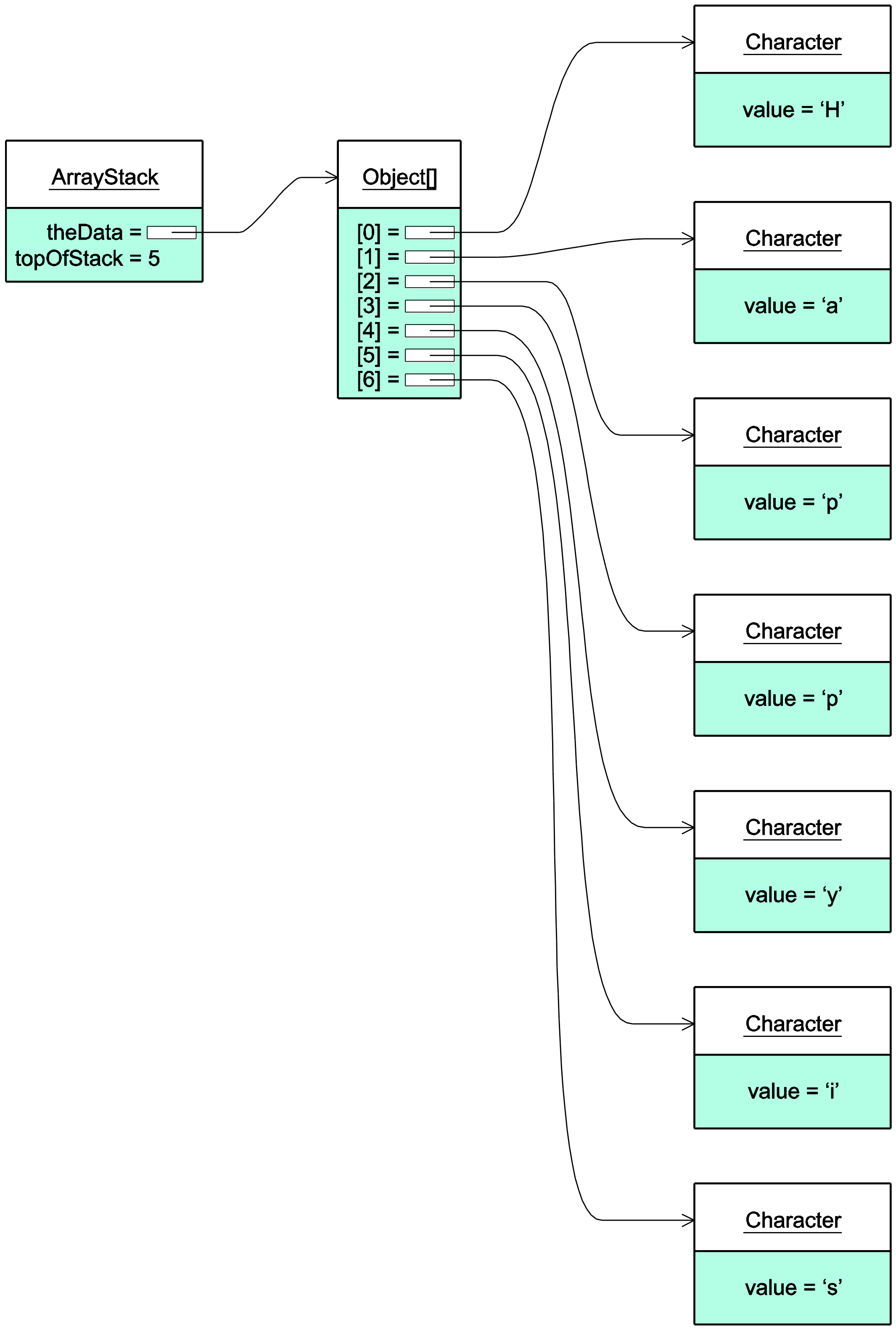
After executing `s.push('i');`



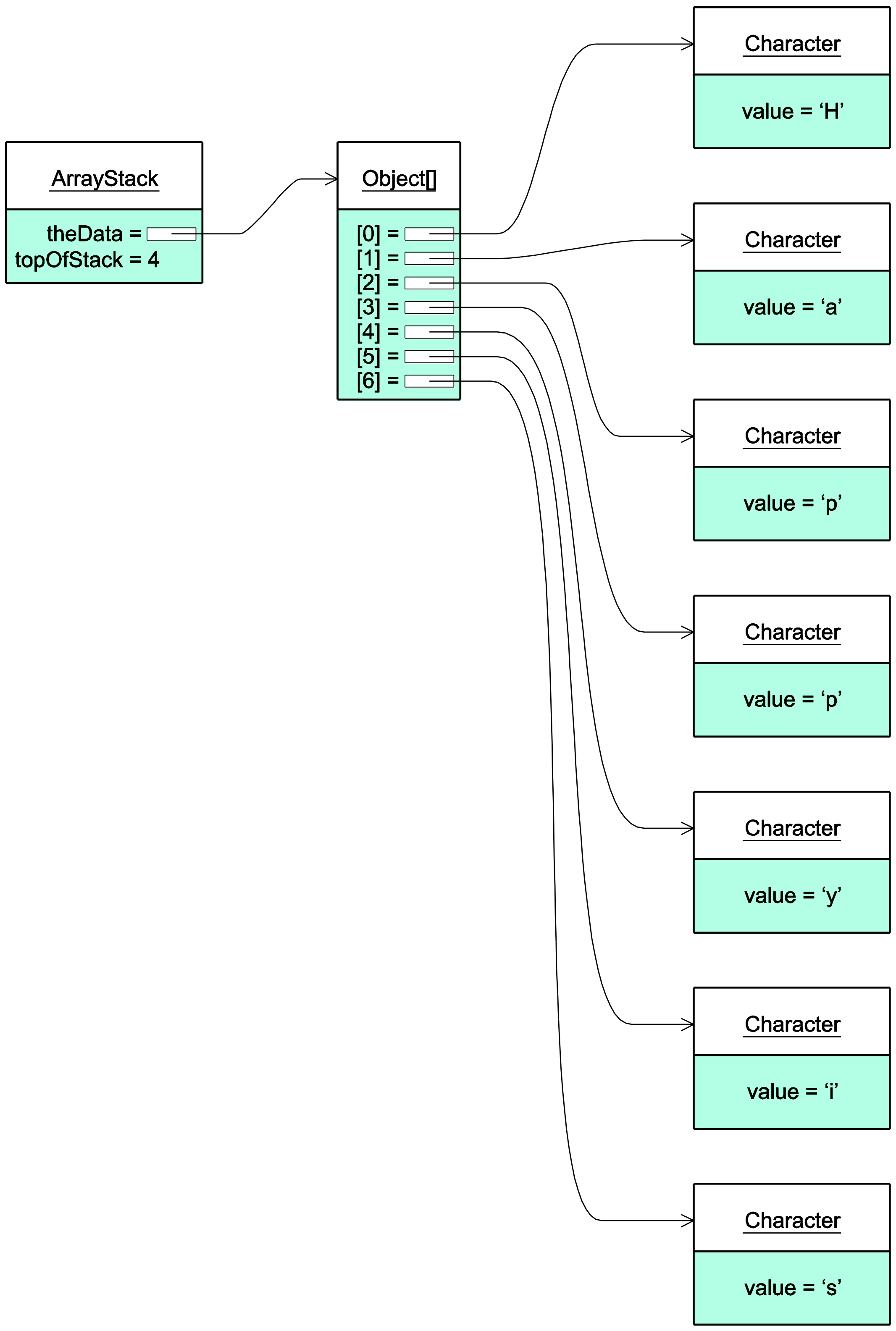
After executing `s.push('s')`



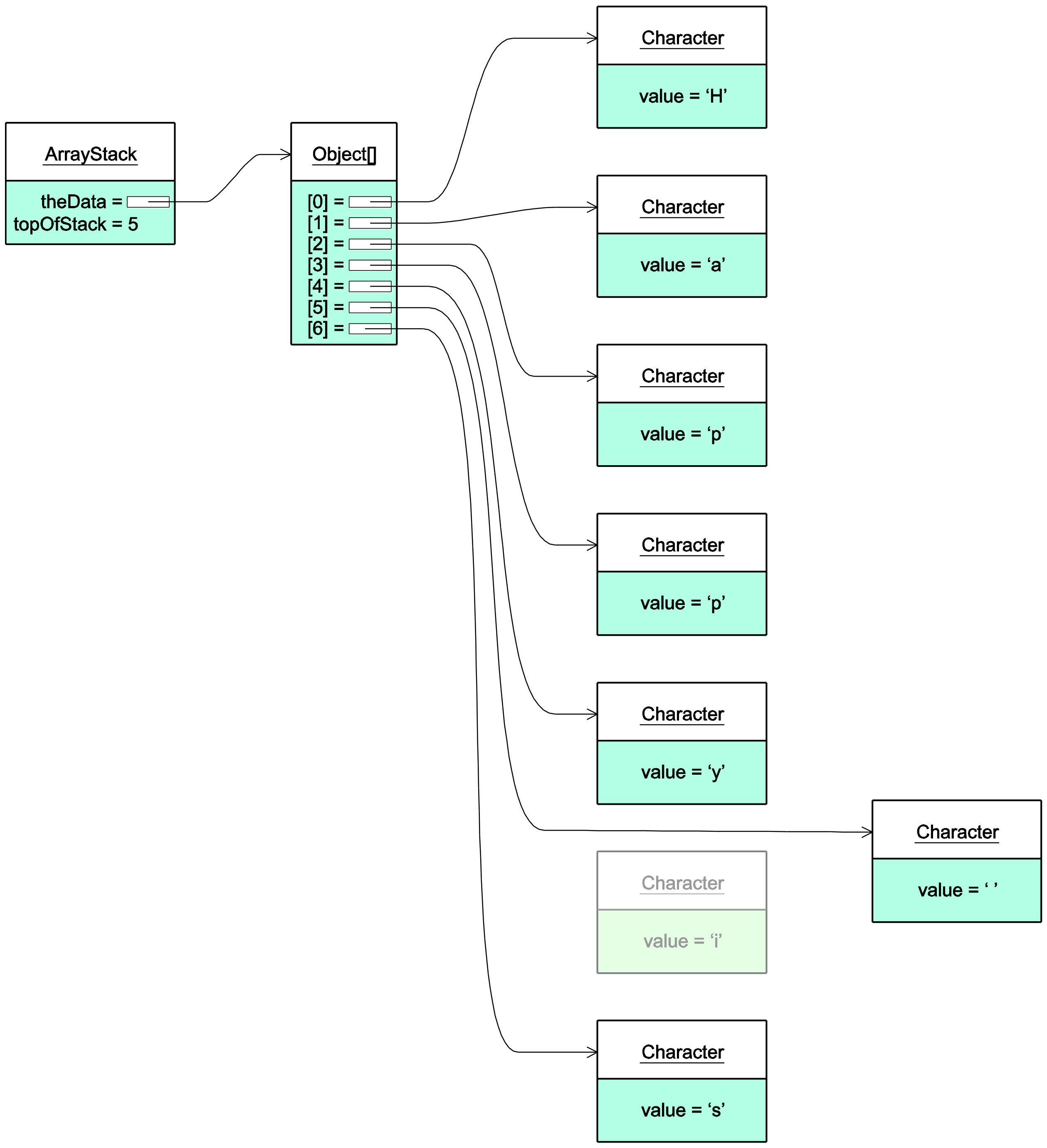
After executing char `ch1 = s.pop();` Note that `ch1=='s'`



After executing `s.pop();`



After executing `s.push(' ');` *Note that the* `Character 'i'` *is shaded to indicate that there are no references to it. It will be garbage collected.*

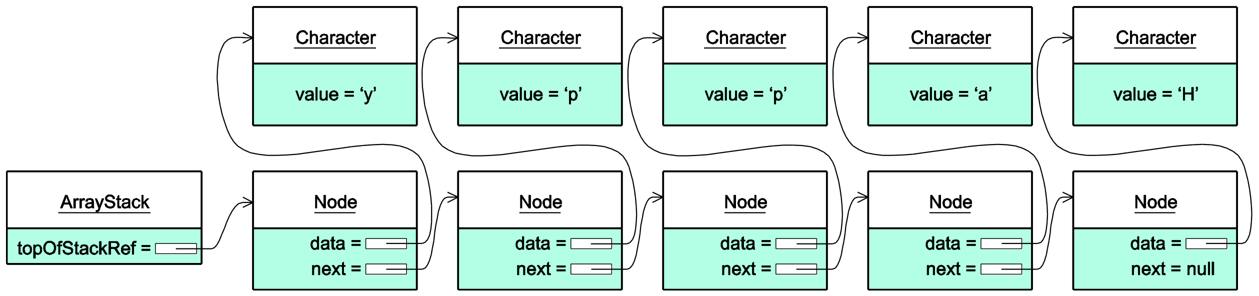


After executing `char ch2 = s.peek(); ch2 == ' '` *and the stack is unchanged.*

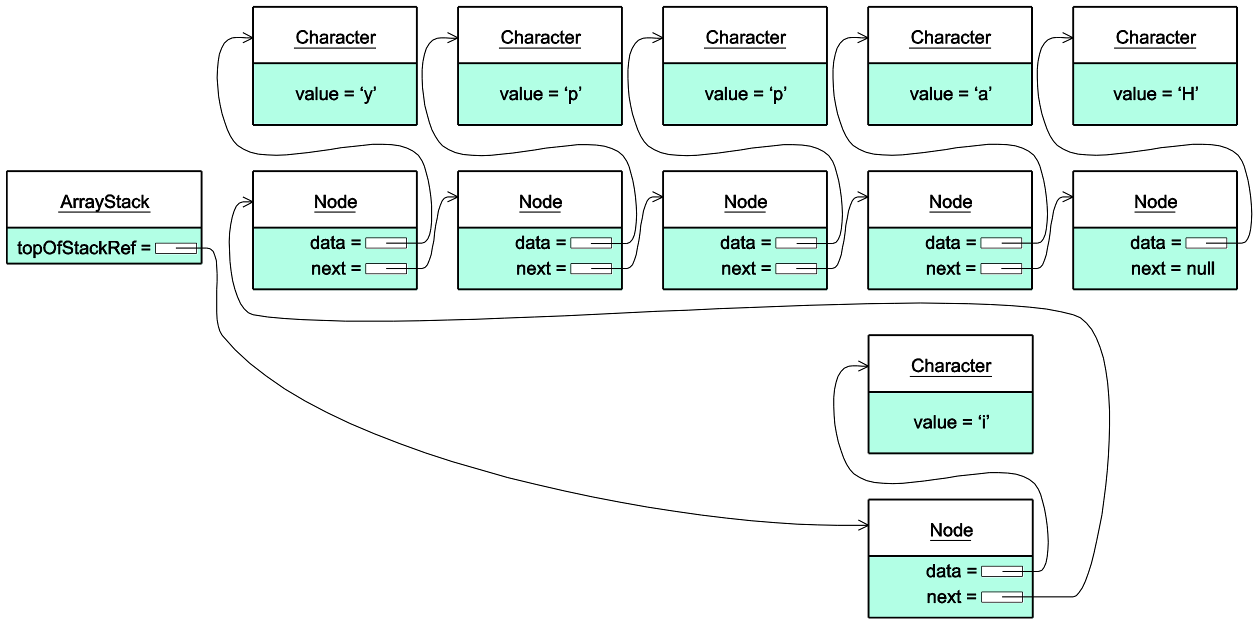
3.3.3

For the implementation of stack s using a linked list of nodes as the underlying data structure (see Figure 3.5), show how the underlying data structure changes after each statement in Question 1 executes. Assume the characters in `"Happy"` are stored on the stack (`H` pushed on first).

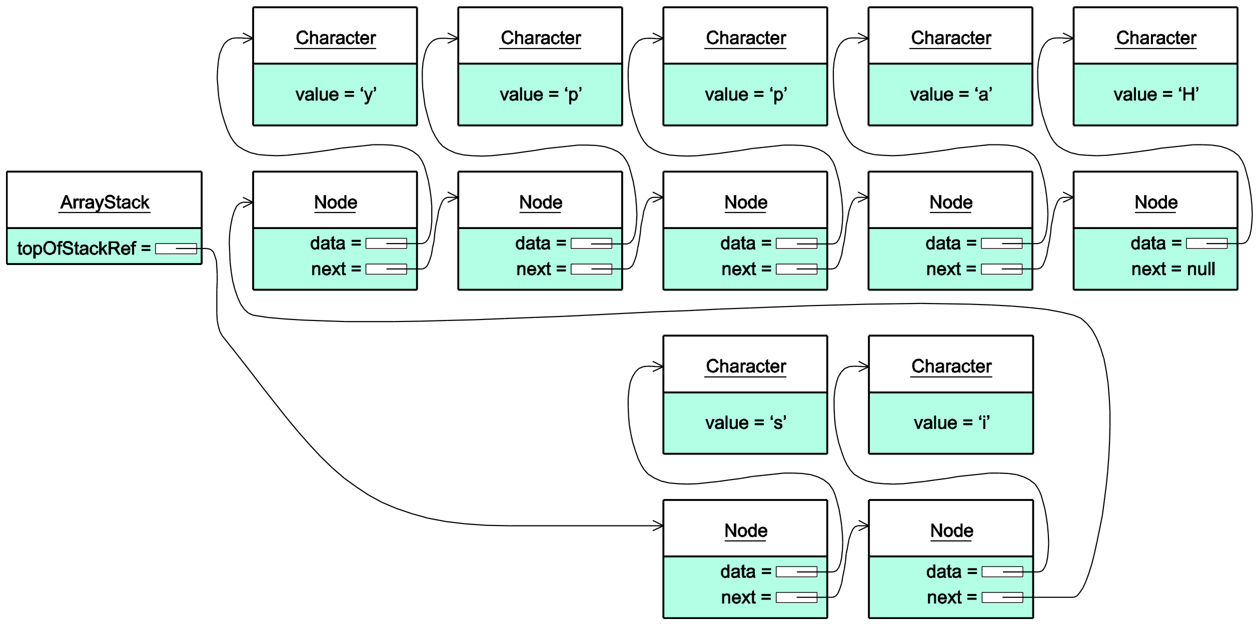
Initial stack:



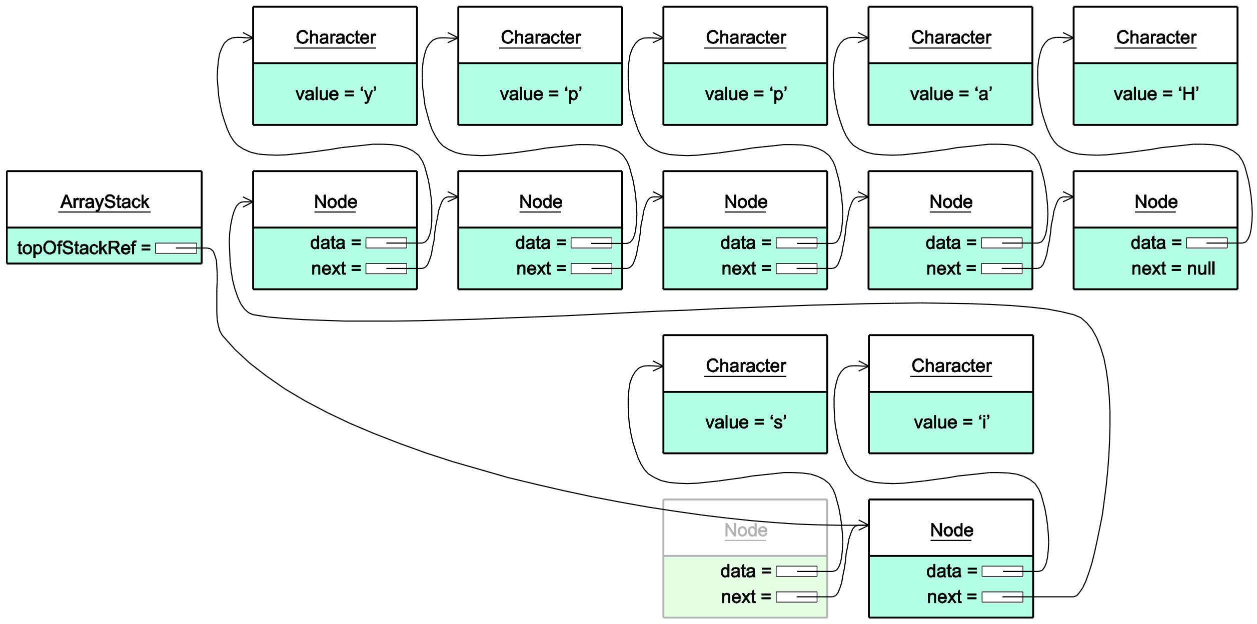
After executing `s.push('i');`



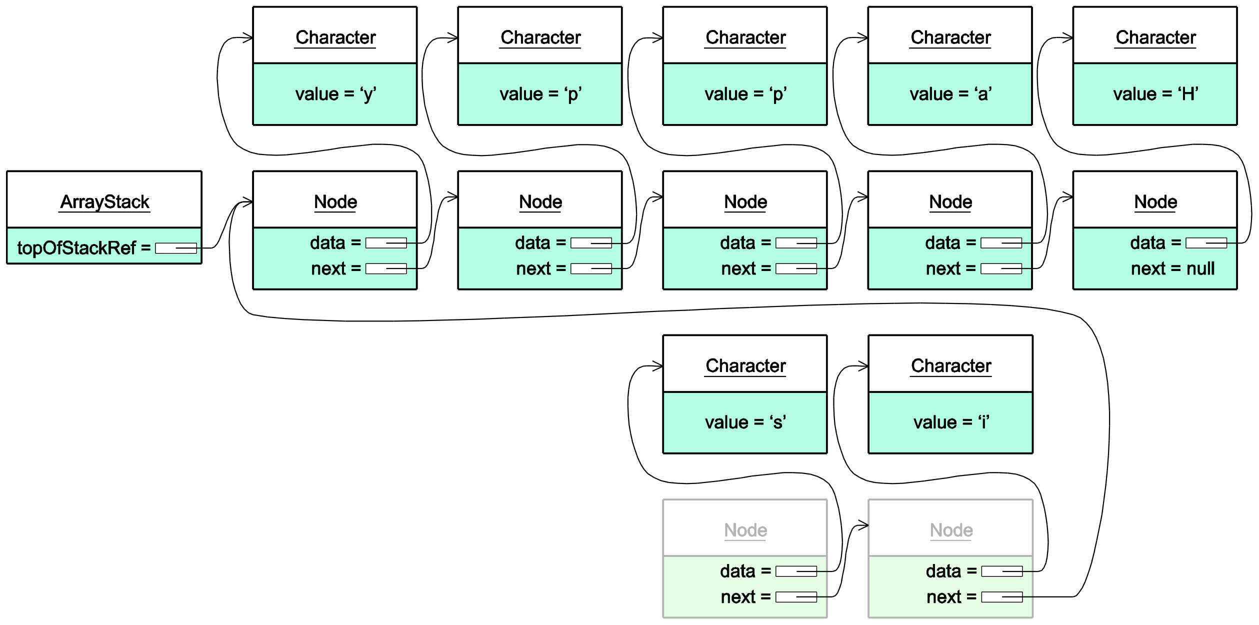
After executing `s.push('s')`



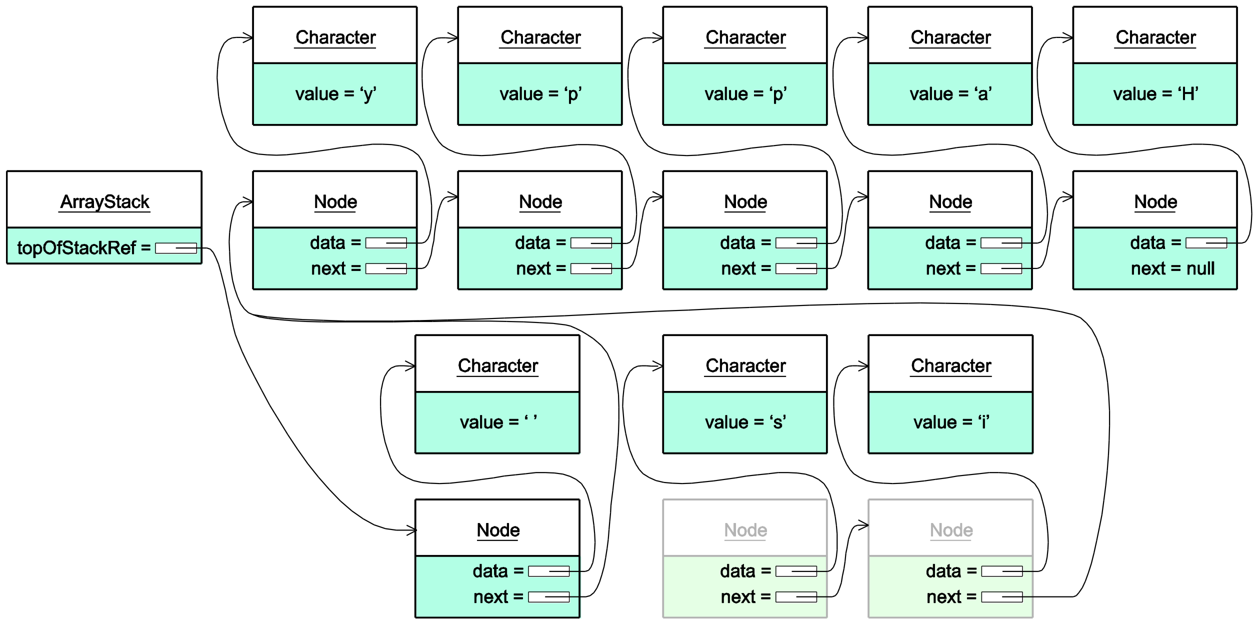
After executing char `ch1 = s.pop();` Note that `ch1=='s'` The Node that references 's' is no longer referenced, so it will be garbage collected.



After executing `s.pop();` The Node that references 's' is no longer referenced, so it will be garbage collected.



After executing `s.push(' ');`



After executing `char ch2 = s.peek(); ch2 == ' '` and the stack is unchanged.

3.4.1

Trace the evaluation of the following expressions using class `PostfixEvaluator`. Show the operand stack each time it is modified.

```
13 2 * 5 / 6 2 5 * - +
```

5 4 * 6 7 + 4 2 / - *

Expression	Action	Stack
13 2 * 5 / 6 2 5 * - + ↑	Push 13	13
13 2 * 5 / 6 2 5 * - + ↑	Push 2	2 13
13 2 * 5 / 6 2 5 * - + ↑	Pop 2 and 13 Evavauate 13 * 2 Push 26	26
13 2 * 5 / 6 2 5 * - + ↑	Push 5	5 26
13 2 * 5 / 6 2 5 * - + ↑	Pop 5 and 26 Evaluate 26/5 Push 5	5
13 2 * 5 / 6 2 5 * - + ↑	Push 6	6 5
13 2 * 5 / 6 2 5 * - + ↑	Push 2	2 6 5
13 2 * 5 / 6 2 5 * - + ↑	Push 5	5 2 6 5
13 2 * 5 / 6 2 5 * - + ↑	Pop 5 and 2 Evaluate 2 * 5 Push 10	10 6 5
13 2 * 5 / 6 2 5 * - + ↑	Pop 10 and 6 Evaluate 6 - 10 Push -4	-4 5
13 2 * 5 / 6 2 5 * - + ↑	Pop -4 and 5 Evaluate 5 + -4 Push 1	1
13 2 * 5 / 6 2 5 * - + ↑	Pop 1 Stack is empty Result is 1	

Expression	Action	Stack
5 4 * 6 7 + 4 2 / - * ↑	Push 5	5
5 4 * 6 7 + 4 2 / - * ↑	Push 4	4 5
5 4 * 6 7 + 4 2 / - * ↑	Pop 4 and 5 Evavauate 5 * 4 Push 20	20
5 4 * 6 7 + 4 2 / - * ↑	Push 6	6 20
5 4 * 6 7 + 4 2 / - * ↑	Push 7	7 5 20
5 4 * 6 7 + 4 2 / - * ↑	Pop 7 and 5 Evaluate 5 + 7 Push 12	12 20
5 4 * 6 7 + 4 2 / - * ↑	Push 4	4 12 20
5 4 * 6 7 + 4 2 / - * ↑	Push 2	2 4 12 20
5 4 * 6 7 + 4 2 / - * ↑	Pop 2 and 4 Evaluate 4 /2 Push 2	2 12 20
5 4 * 6 7 + 4 2 / - * ↑	Pop 2 and 12 Evaluate 12 - 2 Push 10	10 20
5 4 * 6 7 + 4 2 / - * ↑	Pop 10 and 20 Evaluate 20 * 10 Push 200	200
5 4 * 6 7 + 4 2 / - * ↑	Pop 200 Stack is empty Result is 200	