data structures:
lists, stacks, queues

data structures

Data structures:

1. Organize data
2. Enable algorithms on that data
3. Provide book-keeping for algorithms on other data

lists

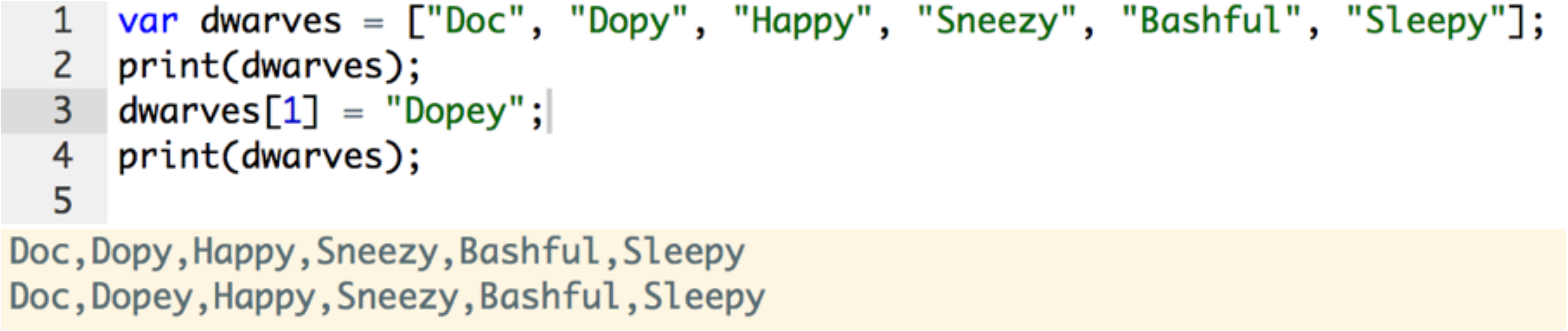
Doc	Dopy	Happy	Sneezy	Bashful	Sleepy
0	1	2	3	4	5

- replace or access at index
- insert, maintaining order
- remove, maintaining order

array: replace

Dopey would like his name spelled correctly.

Doc	Dopy	Happy	Sneezy	Bashful	Sleepy
0	1	2	3	4	5



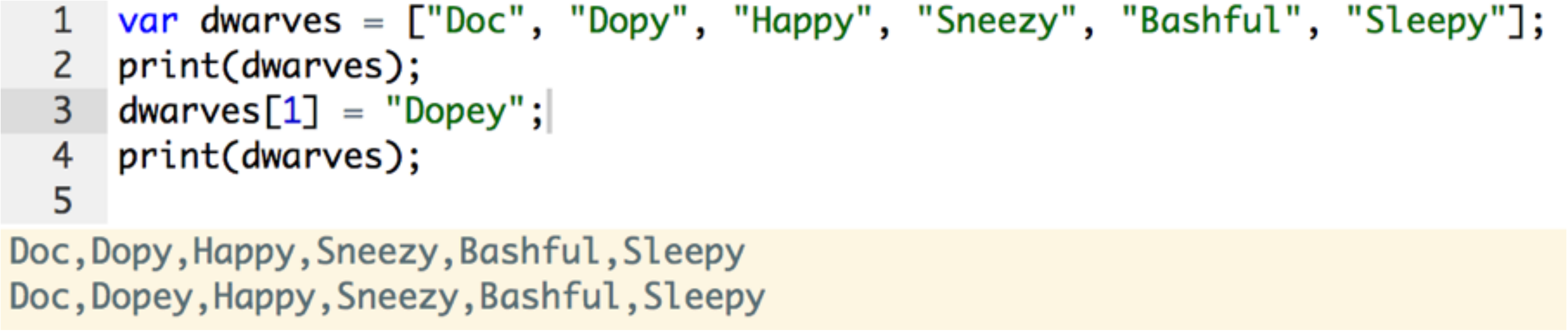
```
var dwarves = ["Doc", "Dopy", "Happy", "Sneezy", "Bashful", "Sleepy"];
print(dwarves);
dwarves[1] = "Dopey";
print(dwarves);

```

```
Doc,Dopy,Happy,Sneezy,Bashful,Sleepy
Doc,Dopey,Happy,Sneezy,Bashful,Sleepy
```

array: replace

Dopey would like his name spelled correctly.

Doc	Dopey	Happy	Sneezy	Bashful	Sleepy
0	1	2	3	4	5



```
var dwarves = ["Doc", "Dopy", "Happy", "Sneezy", "Bashful", "Sleepy"];
print(dwarves);
dwarves[1] = "Dopey";
print(dwarves);

```

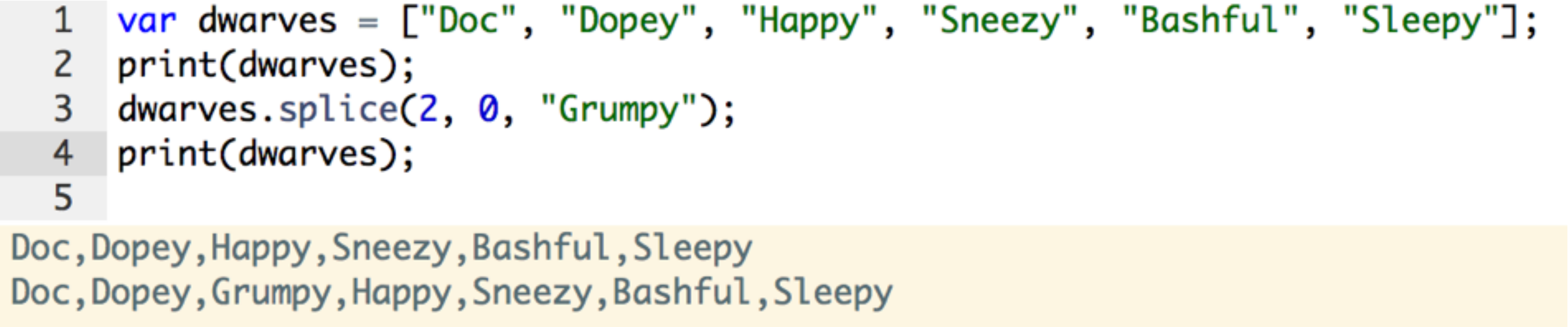
```
Doc,Dopy,Happy,Sneezy,Bashful,Sleepy
Doc,Dopey,Happy,Sneezy,Bashful,Sleepy
```

Time cost: O(1)

array: insert

Grumpy gets in bed between Dopey and Happy

Doc	Dopey	Happy	Sneezy	Bashful	Sleepy
0	1	2	3	4	5



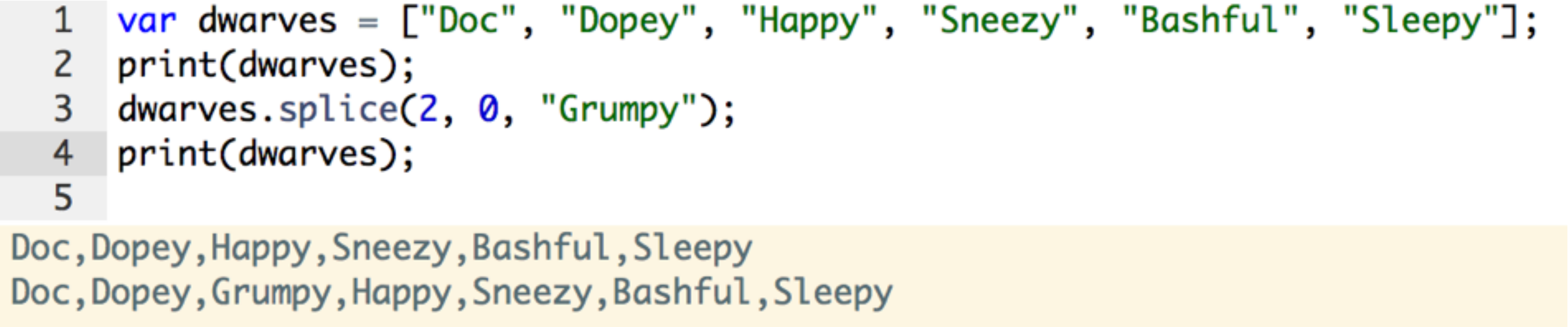
```
var dwarves = ["Doc", "Dopey", "Happy", "Sneezy", "Bashful", "Sleepy"];
print(dwarves);
dwarves.splice(2, 0, "Grumpy");
print(dwarves);

```

```
Doc,Dopey,Happy,Sneezy,Bashful,Sleepy
Doc,Dopey,Grumpy,Happy,Sneezy,Bashful,Sleepy
```

array: insert

Extend the bed (array)

Doc	Dopey	Happy	Sneezy	Bashful	Sleepy	
0	1	2	3	4	5	6



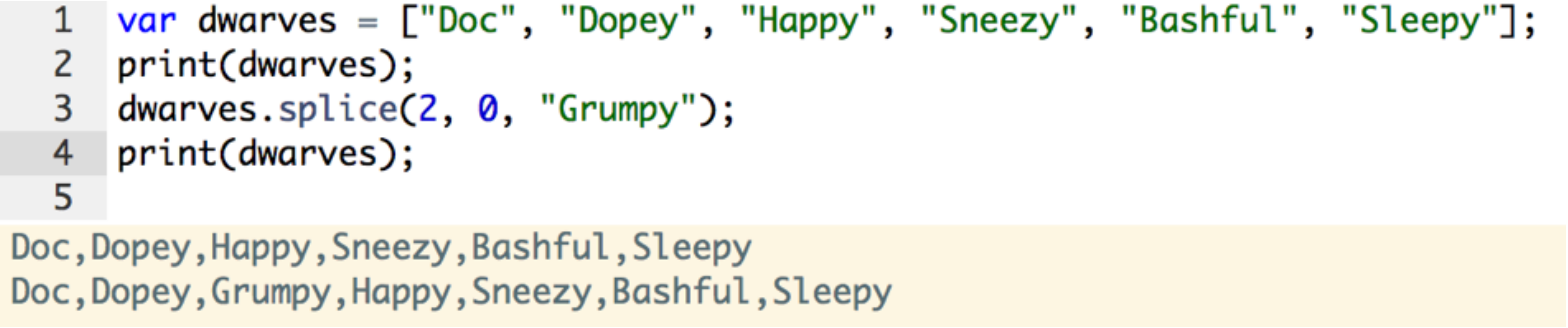
```
var dwarves = ["Doc", "Dopey", "Happy", "Sneezy", "Bashful", "Sleepy"];
print(dwarves);
dwarves.splice(2, 0, "Grumpy");
print(dwarves);

```

```
Doc,Dopey,Happy,Sneezy,Bashful,Sleepy
Doc,Dopey,Grumpy,Happy,Sneezy,Bashful,Sleepy
```

array: insert

Slide items to the right

Doc	Dopey		Happy	Sneezy	Bashful	Sleepy
0	1	2	3	4	5	6



```
var dwarves = ["Doc", "Dopey", "Happy", "Sneezy", "Bashful", "Sleepy"];
print(dwarves);
dwarves.splice(2, 0, "Grumpy");
print(dwarves);

```

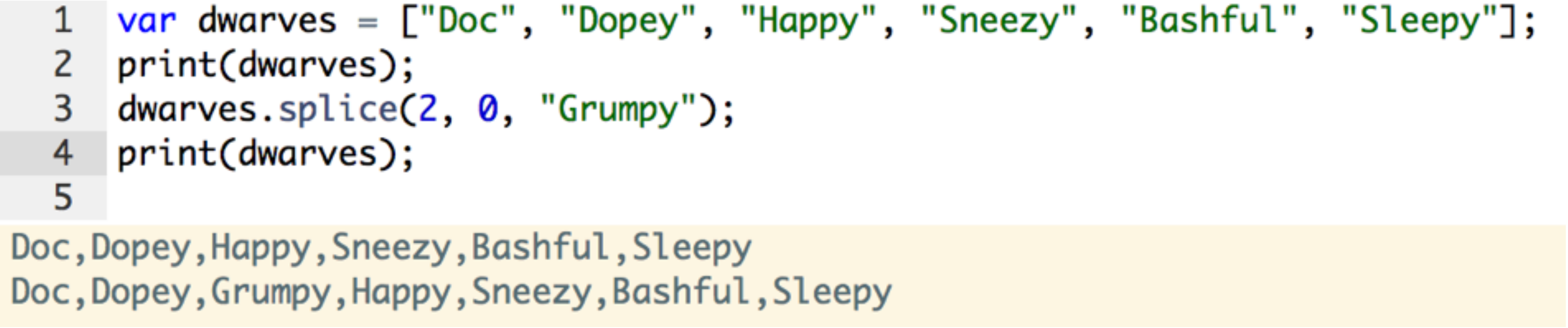
```
Doc,Dopey,Happy,Sneezy,Bashful,Sleepy
Doc,Dopey,Grumpy,Happy,Sneezy,Bashful,Sleepy
```

Time cost: O(n)

array: insert

Copy in “Grumpy”

Doc	Dopey	Grumpy	Happy	Sneezy	Bashful	Sleepy
0	1	2	3	4	5	6



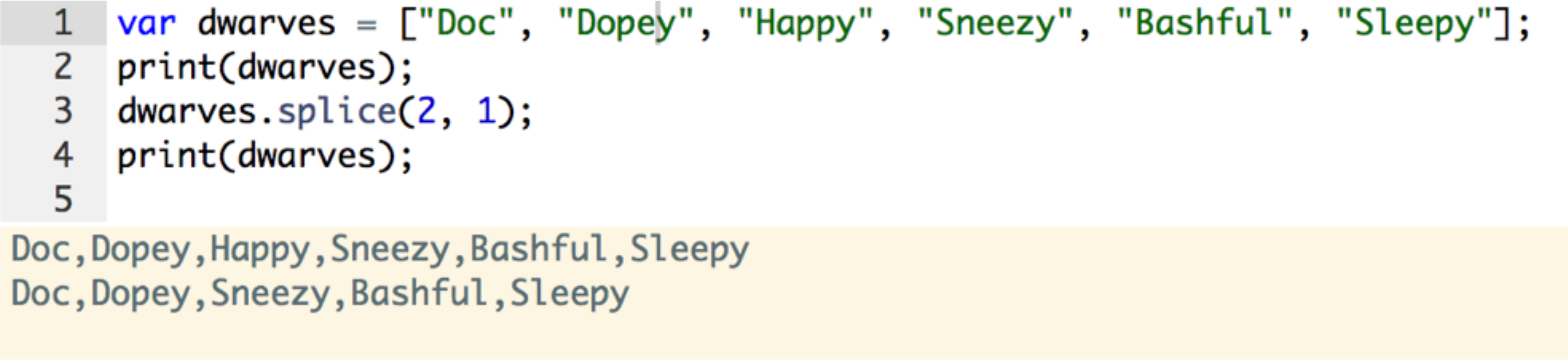
```
var dwarves = ["Doc", "Dopey", "Happy", "Sneezy", "Bashful", "Sleepy"];
print(dwarves);
dwarves.splice(2, 0, "Grumpy");
print(dwarves);

```

```
Doc,Dopey,Happy,Sneezy,Bashful,Sleepy
Doc,Dopey,Grumpy,Happy,Sneezy,Bashful,Sleepy
```

array: remove

Happy gets out of bed and goes to work

Doc	Dopey	Grumpy	Happy	Sneezy	Bashful	Sleepy
0	1	2	3	4	5	6



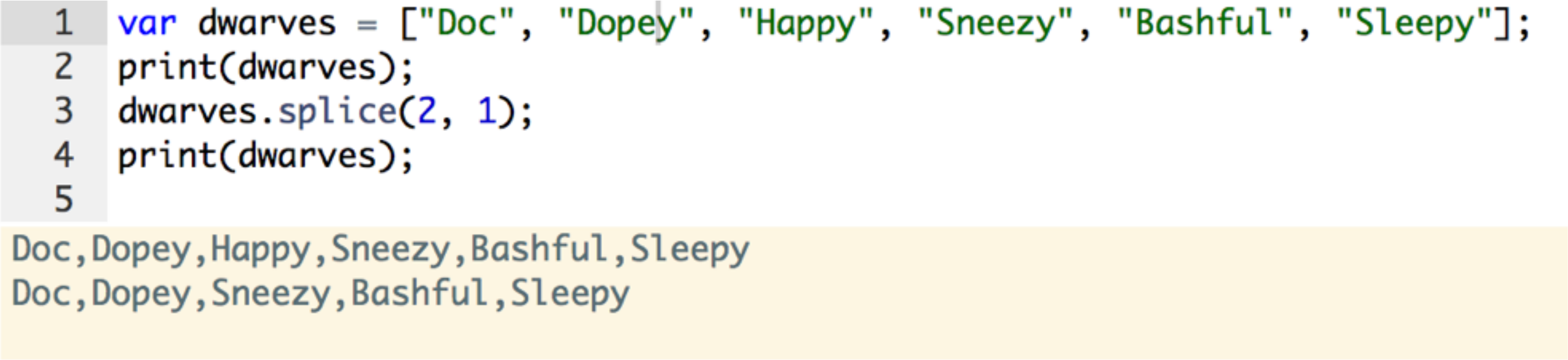
```
var dwarves = ["Doc", "Dopey", "Happy", "Sneezy", "Bashful", "Sleepy"];
print(dwarves);
dwarves.splice(2, 1);
print(dwarves);

```

```
Doc,Dopey,Happy,Sneezy,Bashful,Sleepy
Doc,Dopey,Sneezy,Bashful,Sleepy
```

array: remove

Happy gets out of bed and goes to work

Doc	Dopey	Grumpy		Sneezy	Bashful	Sleepy
0	1	2	3	4	5	6



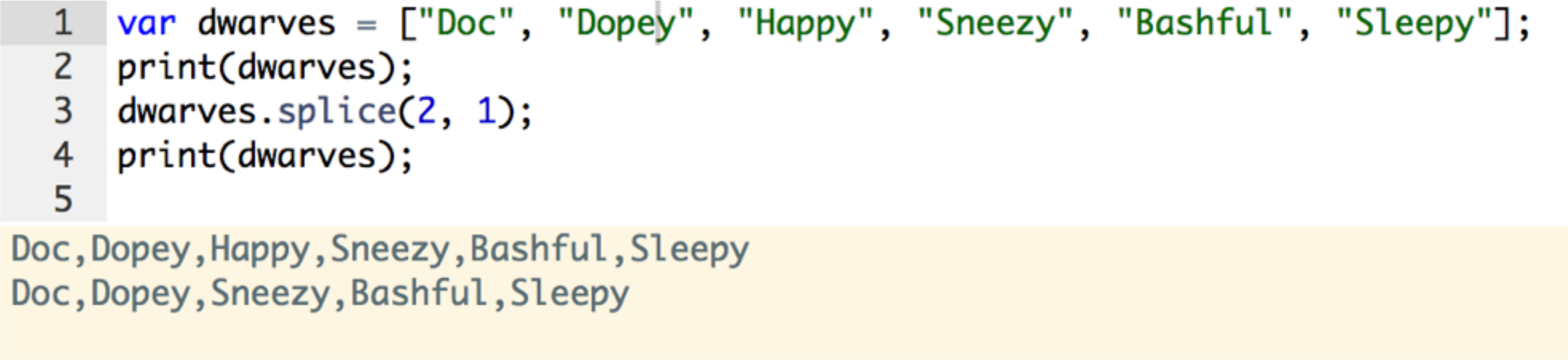
```
var dwarves = ["Doc", "Dopey", "Happy", "Sneezy", "Bashful", "Sleepy"];
print(dwarves);
dwarves.splice(2, 1);
print(dwarves);

```

```
Doc,Dopey,Happy,Sneezy,Bashful,Sleepy
Doc,Dopey,Sneezy,Bashful,Sleepy
```

array: remove

Happy gets out of bed and goes to work

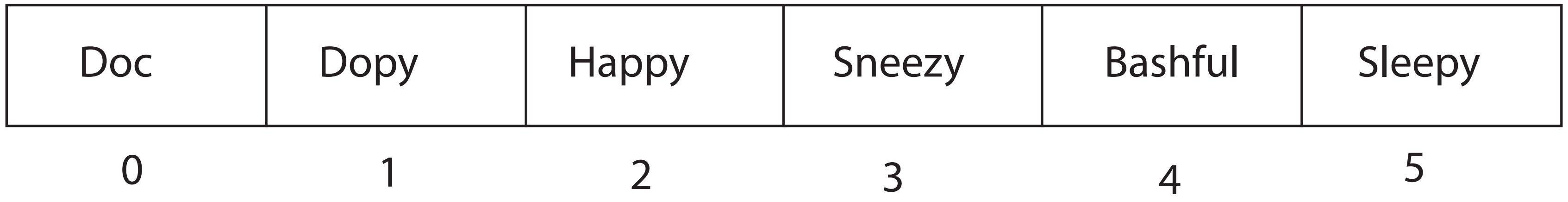
Doc	Dopey	Grumpy	Sneezy	Bashful	Sleepy	
0	1	2	3	4	5	6



```javascript
var dwarves = ["Doc", "Dopey", "Happy", "Sneezy", "Bashful", "Sleepy"];
print(dwarves);
dwarves.splice(2, 1);
print(dwarves);

```

```
Doc,Dopey,Happy,Sneezy,Bashful,Sleepy
Doc,Dopey,Sneezy,Bashful,Sleepy
```

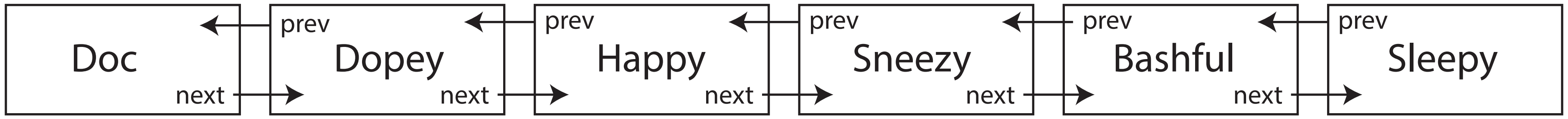
array as list: time costs



Doc	Dopy	Happy	Sneezy	Bashful	Sleepy
0	1	2	3	4	5

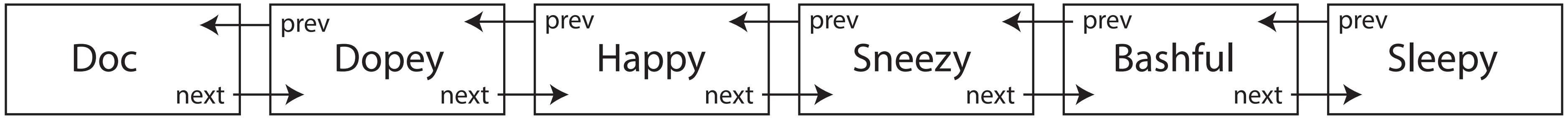
- replace or access at index: O(1)
- insert, maintaining order: O(n)
- remove, maintaining order: O(n)

linked lists



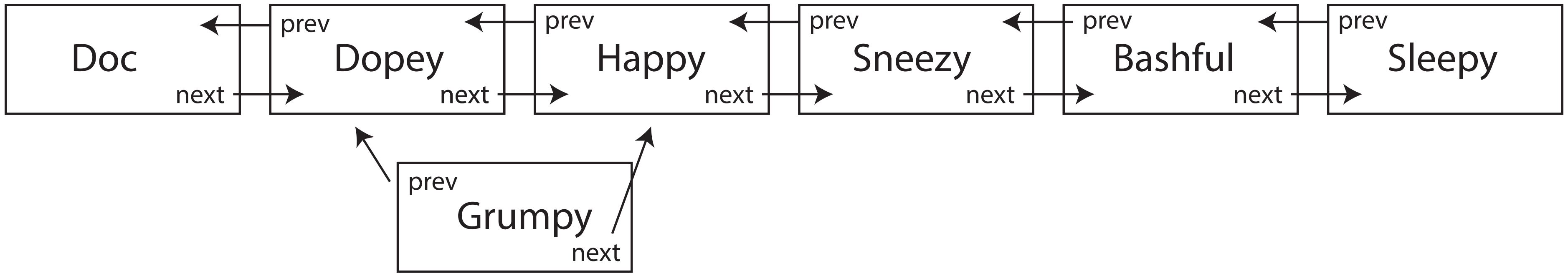
- replace or access at index: O(n)
- insert, maintaining order: O(1)
- remove, maintaining order: O(1)

linked lists: insert



(Student demo with name tags.)

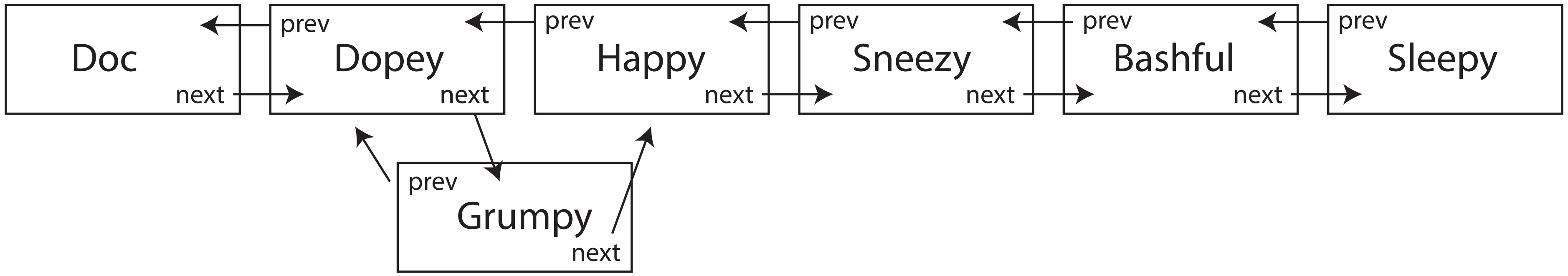
linked lists: insert



1. Create a new node item, with prev and next pointing to predecessor and follower.

Run-time: O(1)

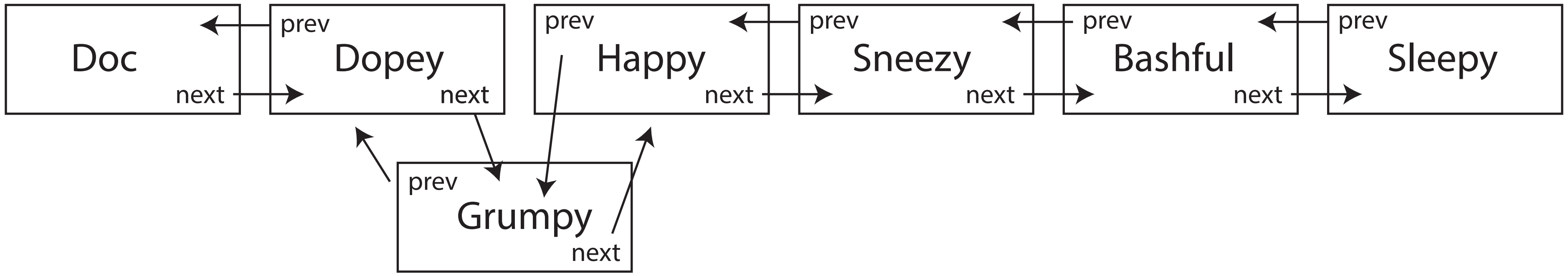
linked lists: insert



2. Update next link of predecessor

Run-time: O(1)

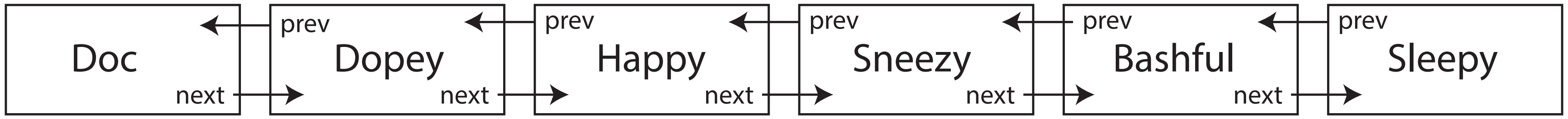
linked lists: insert

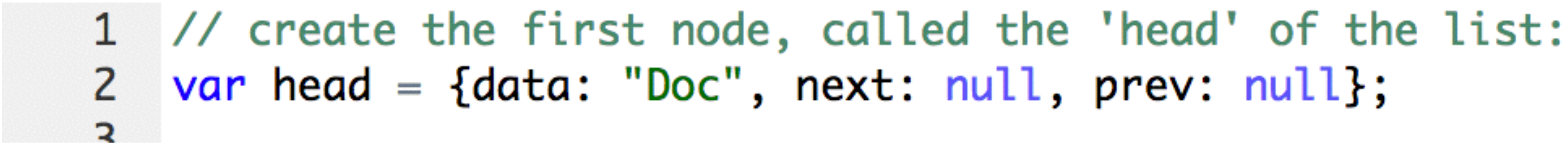


3. Update prev link of follower

Run-time: O(1)

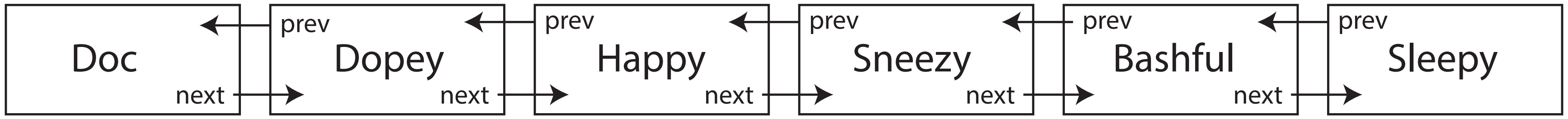
linked lists: creation

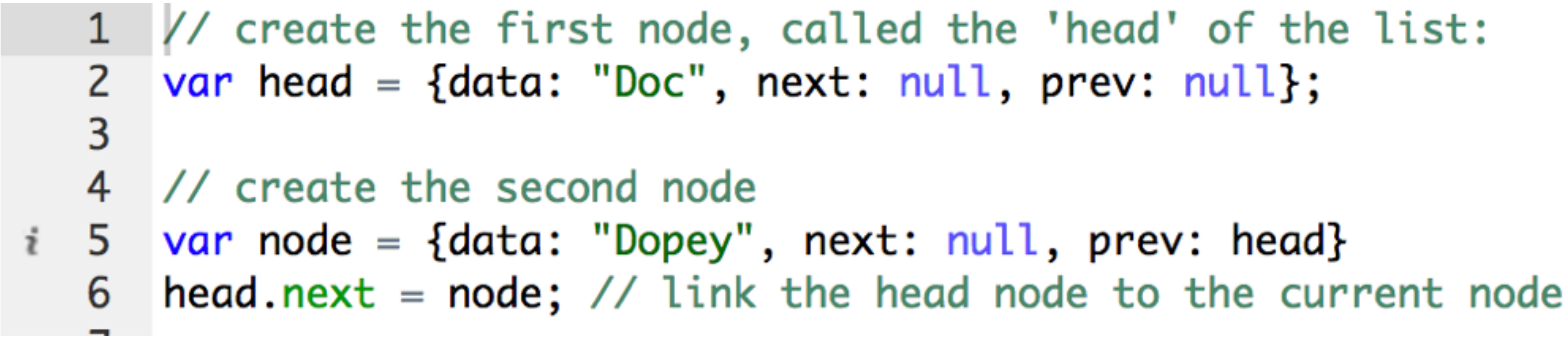




```
// create the first node, called the 'head' of the list:
var head = {data: "Doc", next: null, prev: null};
```

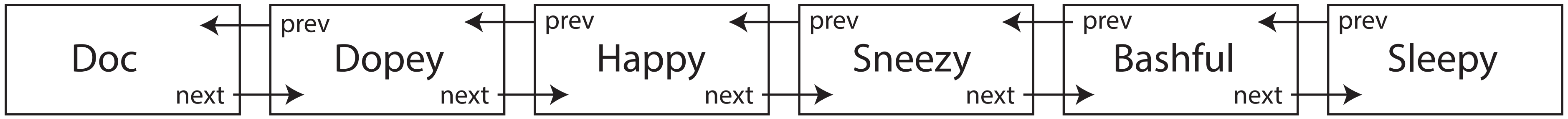
linked lists: creation

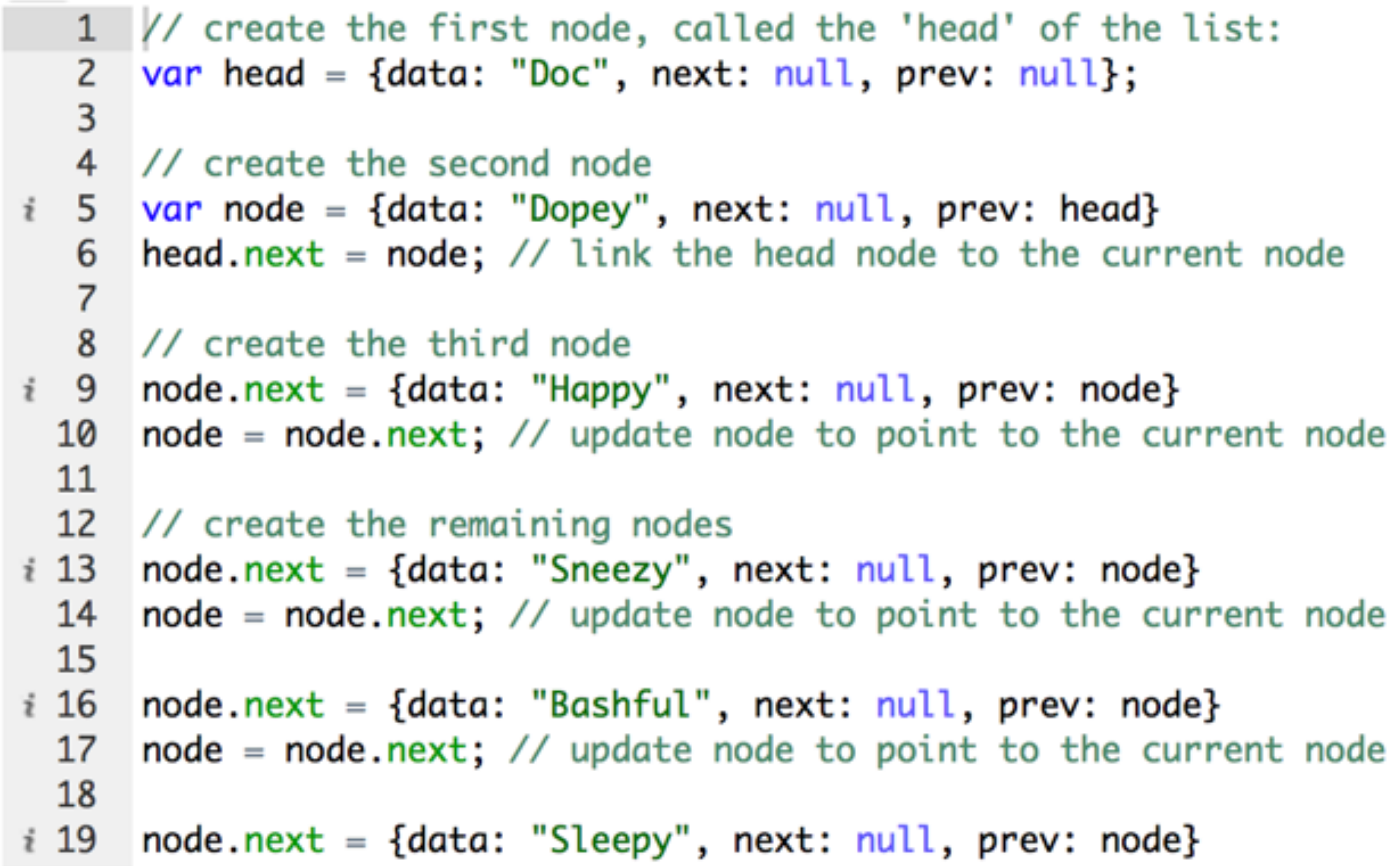




```
// create the first node, called the 'head' of the list:
var head = {data: "Doc", next: null, prev: null};

// create the second node
var node = {data: "Dopey", next: null, prev: head}
head.next = node; // link the head node to the current node
```

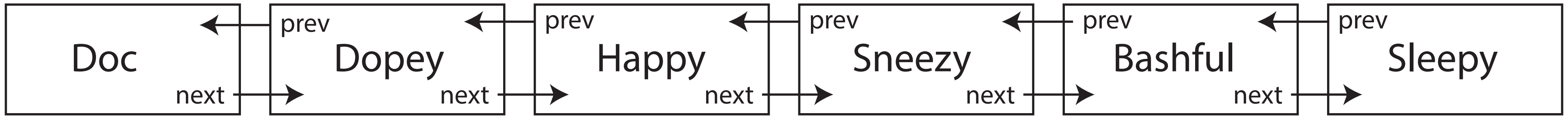
linked lists: creation

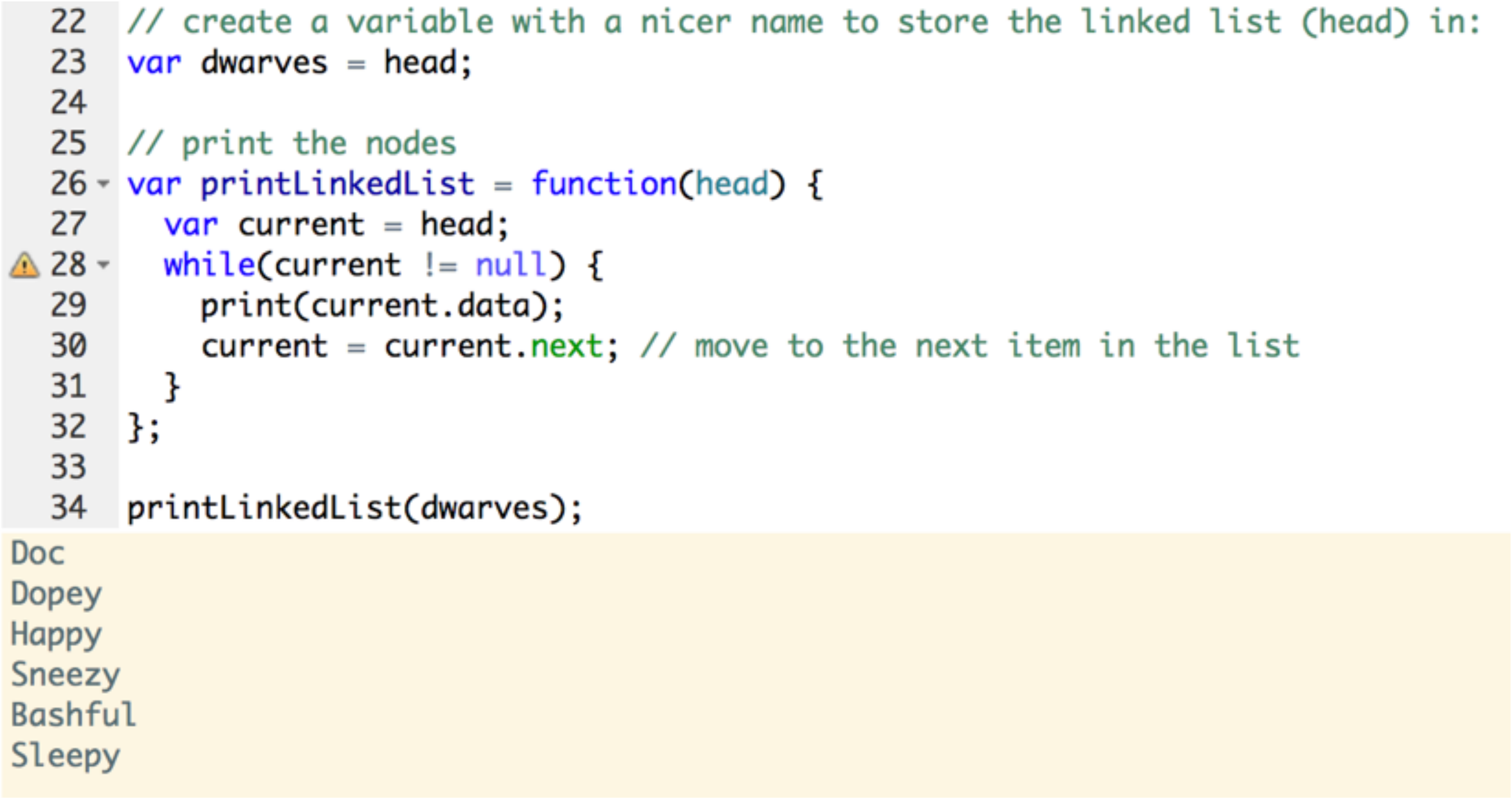




```
// create the first node, called the 'head' of the list:
var head = {data: "Doc", next: null, prev: null};

// create the second node
var node = {data: "Dopey", next: null, prev: head}
head.next = node; // link the head node to the current node

// create the third node
node.next = {data: "Happy", next: null, prev: node}
node = node.next; // update node to point to the current node

// create the remaining nodes
node.next = {data: "Sneezy", next: null, prev: node}
node = node.next; // update node to point to the current node

node.next = {data: "Bashful", next: null, prev: node}
node = node.next; // update node to point to the current node

node.next = {data: "Sleepy", next: null, prev: node}
```

linked lists: looping over



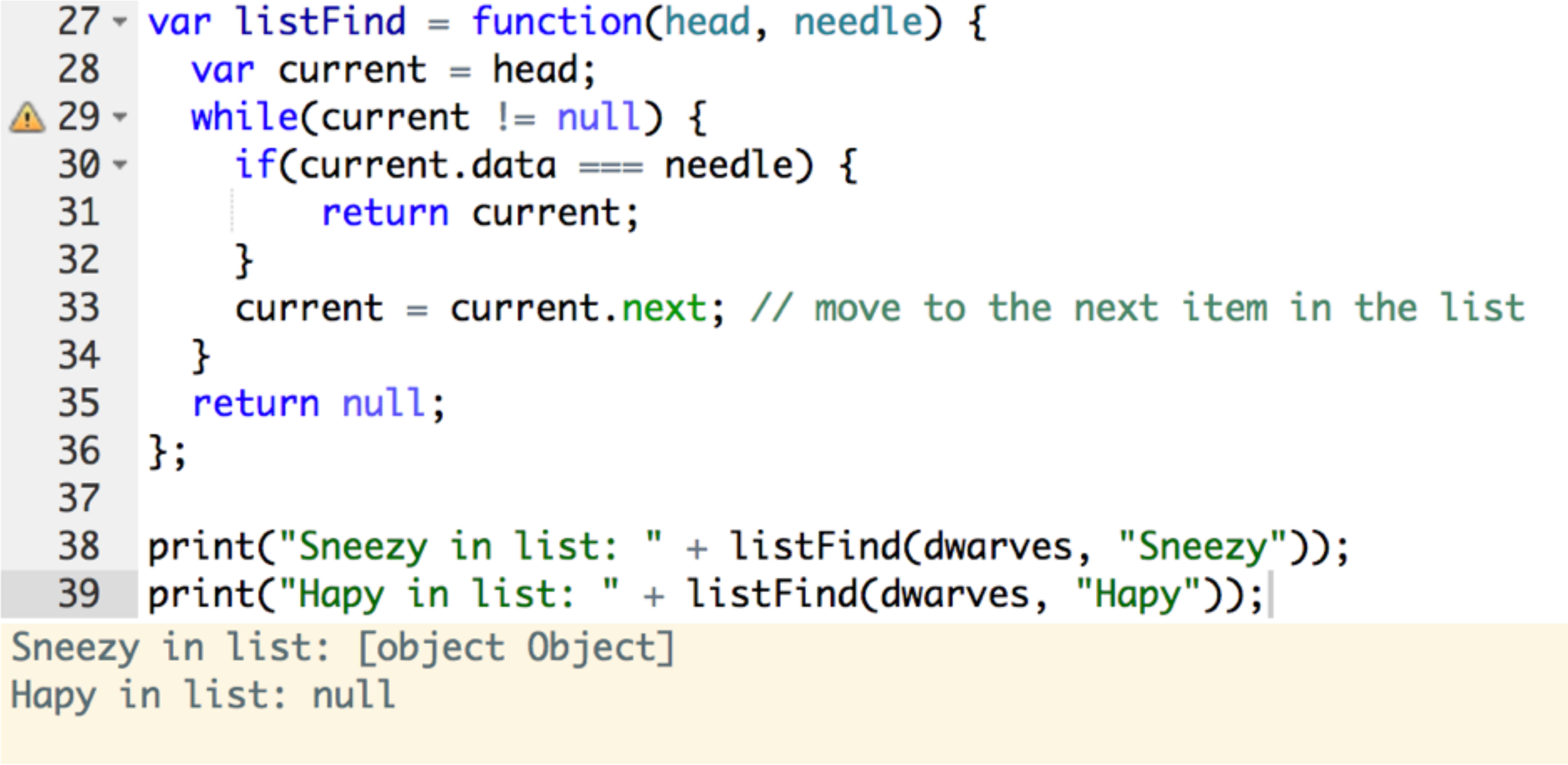


```
// create a variable with a nicer name to store the linked list (head) in:
var dwarves = head;

// print the nodes
var printLinkedList = function(head) {
  var current = head;
  while(current != null) {
    print(current.data);
    current = current.next; // move to the next item in the list
  }
};

printLinkedList(dwarves);
```

```
Doc
Dopey
Happy
Sneezy
Bashful
Sleepy
```

linked lists **exercise: needle in a haystack**

linked lists **exercise: needle in a haystack**



```
var listFind = function(head, needle) {
  var current = head;
  while(current != null) {
    if(current.data === needle) {
        return current;
    }
    current = current.next; // move to the next item in the list
  }
  return null;
};

print("Sneezy in list: " + listFind(dwarves, "Sneezy"));
print("Hapy in list: " + listFind(dwarves, "Hapy"));
```

```
Sneezy in list: [object Object]
Hapy in list: null
```

linked lists **exercise at home: deletion**

Can you write a function that deletes a node from a linked list? (Start by finding the node using the function you already wrote.)

What’s the run-time?

linked lists: why do we care?

Time costs are different, but both arrays and linked lists provide the same operations:

- replace or access at index
- insert, maintaining order
- remove, maintaining order

1. Performance: Maybe you’d use a linked list to represent buckets in a dictionary, or a genome sequence. (Fast deletion or insertion, no indexing.)
2. Understanding: Linked list representation is similar to representation of graphs and networks.

arrays vs linked lists: the list Abstract Data Type

Time costs are different, but both arrays and linked lists provide the same operations:

- replace or access at index
- insert, maintaining order
- remove, maintaining order

If we are describing some other algorithm that uses a list for book-keeping, we don’t want to get into the details.

A list is an **Abstract Data Type** providing certain operations on ordered data.

abstract data types

- ordered list: insert, delete wherever
- stack: insert at the end (top), remove from end
- queue: insert at end, remove from beginning

Stacks model Last-In-First-Out (LIFO) situations
Queues model First-In-First-Out (FIFO) situations

abstract data types

- ordered list: insert, delete wherever
- stack: insert at the end (top), remove from end
- queue: insert at end, remove from beginning

Stacks model Last-In-First-Out (LIFO) situations
Queues model First-In-First-Out (FIFO) situations

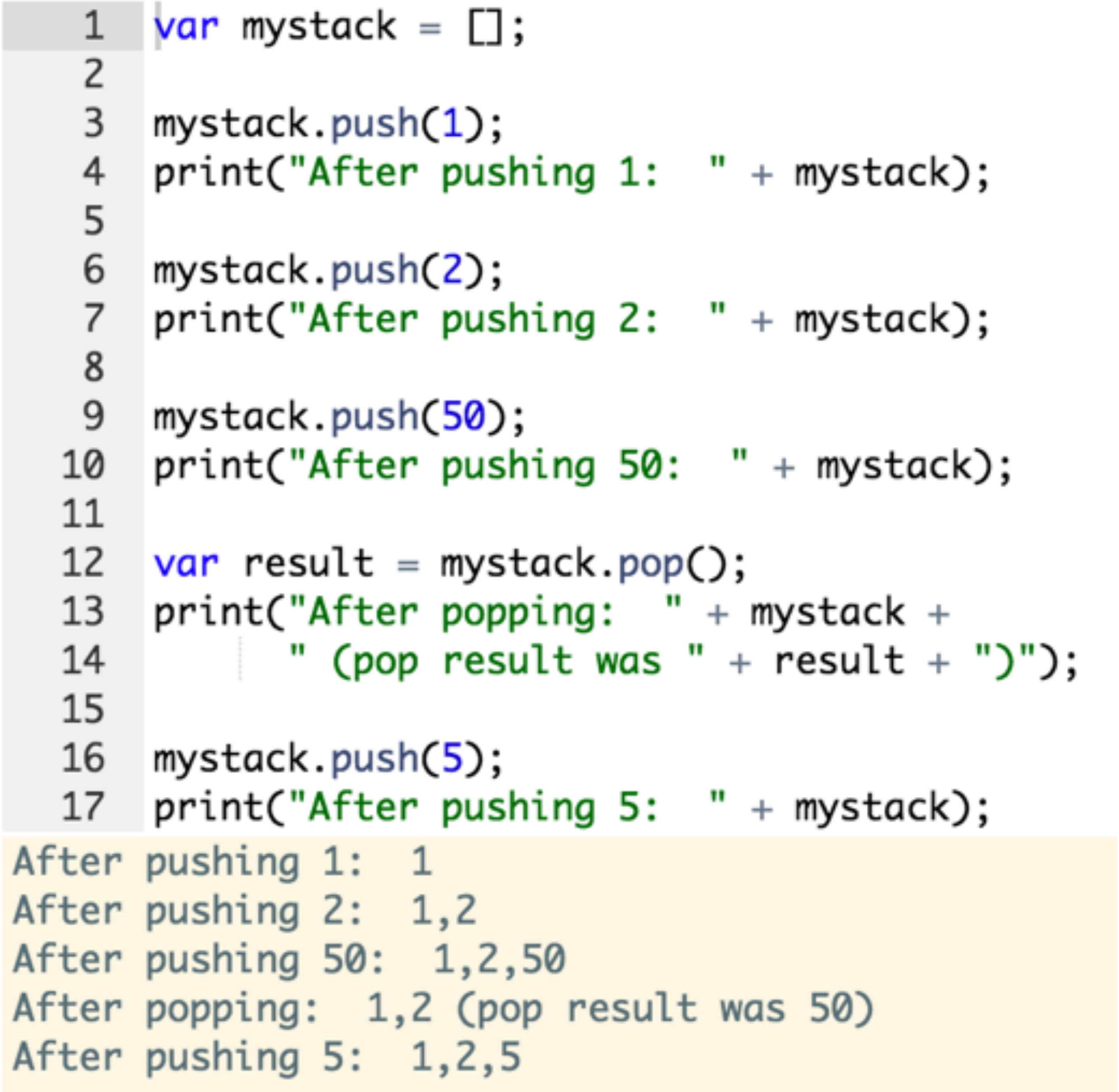
stack: student demo with name tags

stack: implementation

Methods:

- push(): add new item
- pop(): remove most-recently-added item

You can just use a Javascript array.



```
var mystack = [];

mystack.push(1);
print("After pushing 1:  " + mystack);

mystack.push(2);
print("After pushing 2:  " + mystack);

mystack.push(50);
print("After pushing 50:  " + mystack);

var result = mystack.pop();
print("After popping:  " + mystack +
      " (pop result was " + result + ")");

mystack.push(5);
print("After pushing 5:  " + mystack);
```

```
After pushing 1:  1
After pushing 2:  1,2
After pushing 50:  1,2,50
After popping:  1,2 (pop result was 50)
After pushing 5:  1,2,5
```

stack example algorithm: computing expressions

How would you write your own interpreter for a new programming language, TuckScript?

One piece: handling expressions:

“5 + 4 (3 + 2 / 4 * (96 / 2))”

1. Break it apart into symbols (parsing)
2. Apply operator symbols to value symbols
3. Use new values with operators

parentheses and order of operations make it tricky!

stack example algorithm: computing expressions

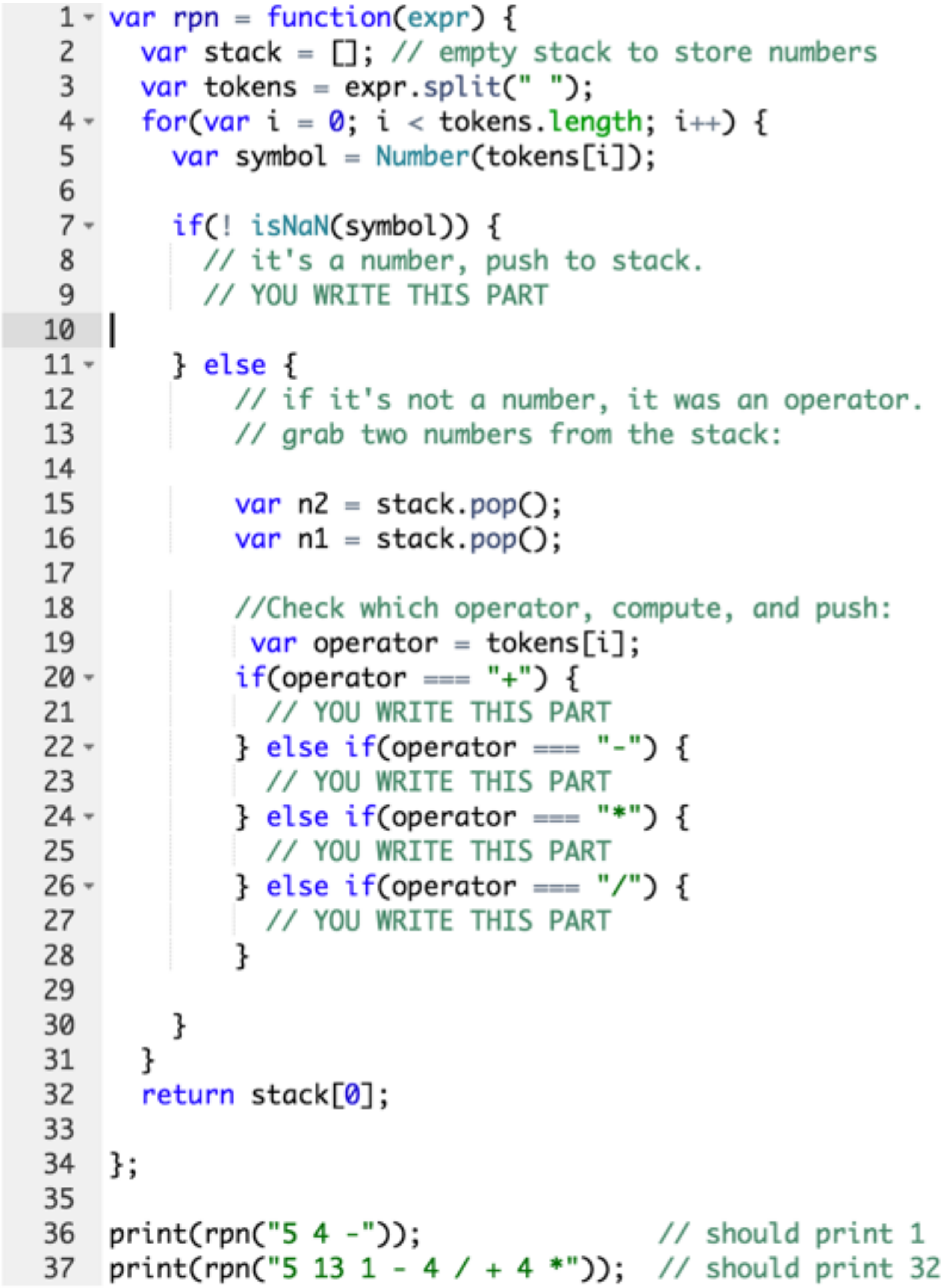
reverse polish notation (RPN):

“3 4 2 * +”

1. Remember 3
2. Remember 4
3. Remember 2
4. *: multiply the last two numbers and replace
5. +: add last two numbers

We can implement “remember” by pushing onto a stack.
(**Visualization** by Daniel Shanker in notes.)

stack **exercise**: writing your own interpreter



```
var rpn = function(expr) {
  var stack = []; // empty stack to store numbers
  var tokens = expr.split(" ");
  for(var i = 0; i < tokens.length; i++) {
    var symbol = Number(tokens[i]);

    if(! isNaN(symbol)) {
      // it's a number, push to stack.
      // YOU WRITE THIS PART

    } else {
        // if it's not a number, it was an operator.
        // grab two numbers from the stack:

        var n2 = stack.pop();
        var n1 = stack.pop();

        //Check which operator, compute, and push:
         var operator = tokens[i];
        if(operator === "+") {
          // YOU WRITE THIS PART
        } else if(operator === "-") {
          // YOU WRITE THIS PART
        } else if(operator === "*") {
          // YOU WRITE THIS PART
        } else if(operator === "/") {
          // YOU WRITE THIS PART
        }

    }
  }
  return stack[0];

};

print(rpn("5 4 -"));               // should print 1
print(rpn("5 13 1 - 4 / + 4 *"));  // should print 32
```

stack: applications

- keeping track of running and suspended functions
- parsing code for compilers or interpreters (together with function table)
- decision making algorithms (e.g. searching a maze using hand-on-wall rule.)

queue

- ordered list: insert, delete wherever
- stack: insert at the end (top), remove at end
- queue: insert at end, remove from beginning

Stacks model Last-In-First-Out (LIFO) situations
Queues model First-In-First-Out (FIFO) situations

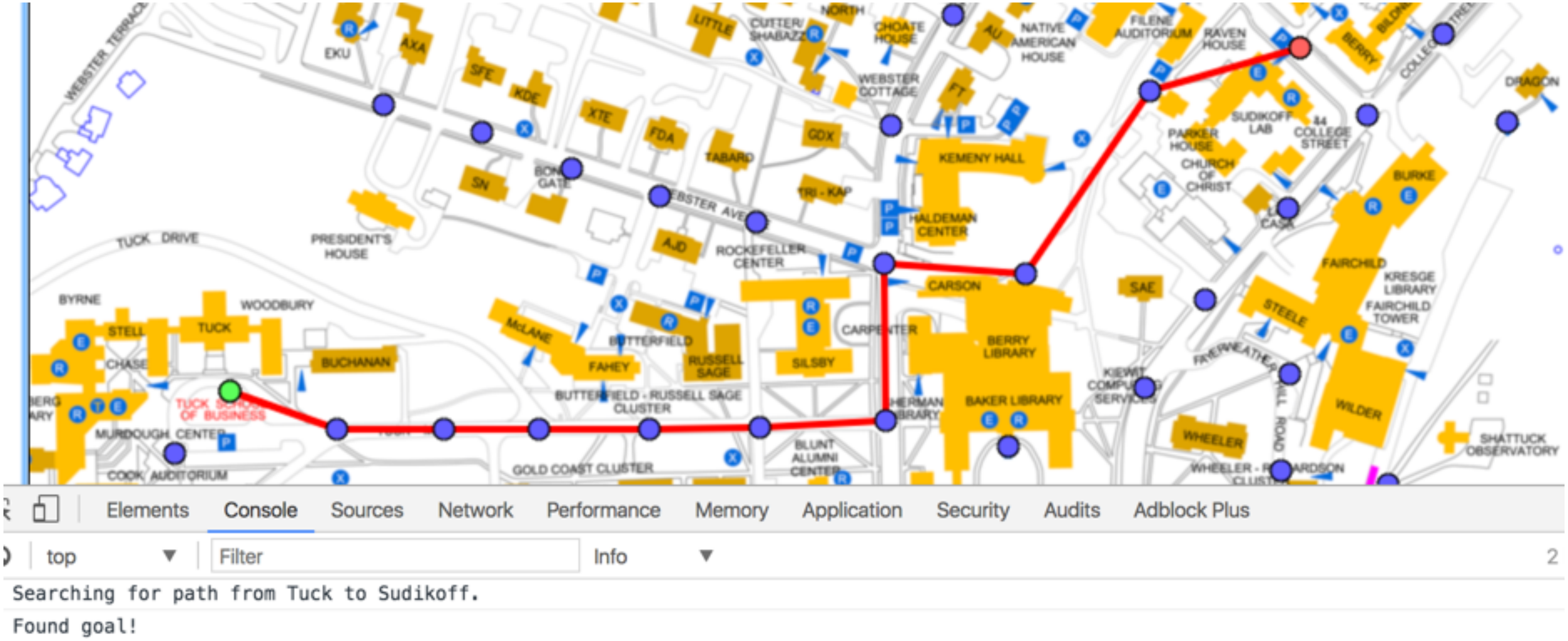
queue: student demo with name tags

queue: implementation

1. Use javascript array, with push() and shift()?
    - theoretical runtime bad
    - I used for assignment 4
2. Linked list
    - theoretically better run-time
    - probably very slow in javascript, not built-in

queue: applications

- Swapping between processes running on CPU
- Handling server requests in order
- Graphs and graph-search algorithms for ordered exploration and decision making.
- packet handling: ethernet, internet, multi-core



```
Searching for path from Tuck to Sudikoff.
Found goal!
```

Doc

Dopey

Grumpy

Happy

Sneezy

Bashful

Sleepy