

Lecture 31: Efficiency of list operations

This lecture and part of the next explain how lists can be implemented in computer memory, the box-and-pointer representation of lists, including the ideas of cons cell, pointer, shared substructure, loops of pointers, and dotted pairs, and the Scheme procedures `set-car!`, `set-cdr!`, `eq?`, and `pair?` As an example, we describe an efficient implementation of queues.

Implementing lists

Suppose we execute the command:

```
(define x '(2 4))
```

Then the value of `x` is the Scheme list `(2 4)`. How could this list be represented in computer memory? A traditional choice is to allocate two consecutive memory words for each element of the list, one to point to the element, and one to point to the rest of the list. The pairs of cells for different elements need not be contiguous in memory. For example, we might have the following representation of the list `(2 4)`:

address:	contents
93:	4
100:	420
101:	306
306:	93
307:	-1
420:	2

Here the memory address 100 is a pointer to the list, which is the value of `x`. The pair of cells at 100 and 101 indicate that the first element of the list is to be found at address 420, which has contents 2, and that the rest of the list is to be found at address 306. The pair of cells at 306 and 307 indicate that the second element of the list is to be found at address 93, which has contents 4, and that the rest of the list is the null list, indicated by the value -1. The addresses 100, 420, 306, 93 are functioning as pointers, that is, values that give the locations of other values. To make this really work, we'd have to add enough information to our representation to be able to tell whether a pointer was pointing to a basic data value or to another list structure.

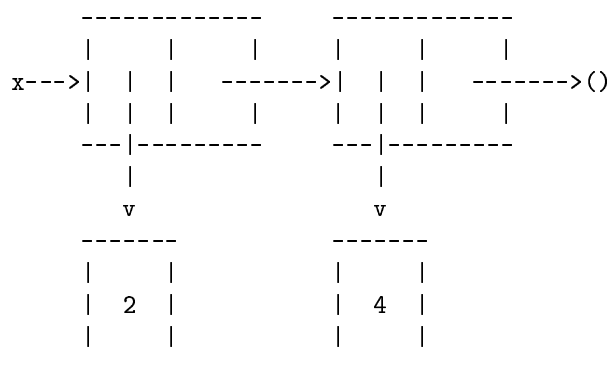
With this representation of lists, the basic operations of `cons`, `car`, and `cdr` can each be implemented in time $O(1)$, that is, constant time independent of the size of the lists involved. To see this, note that the value of `x` is the pointer 100. To take the `car` of `x`, we get the contents of address 100, which is the pointer 420, which points to the value 2. To take the `cdr` of `x`, we get the contents of address 101, which is the pointer 306, which points to the list `(4)`. To `cons` the number 3 onto the list `x`, we allocate (if necessary) a memory word to hold 3, say at address 450, and allocate a pair of memory words, say at addresses 460 and 461, and put the pointer to 450 into address 460, and the value of `x` into address 461. The value returned would be the pointer 460, which points to the list `(3 2 4)`. The result would be the following memory contents.

address:	contents
93:	4
100:	420
101:	306
306:	93
307:	-1
420:	2
450:	3
460:	450
461:	100

Note that each of these operations involved allocating, copying, or changing a bounded number of memory locations, independent of the length of the list.

Box and pointer diagrams

Abstracting away from the specific addresses, and representing the pairs of memory cells by two side-by-side boxes, called a cons cell, and the pointers by arrows, we can give a box-and-pointer representation of the fact that the value of `x` is the list `(2 4)`.

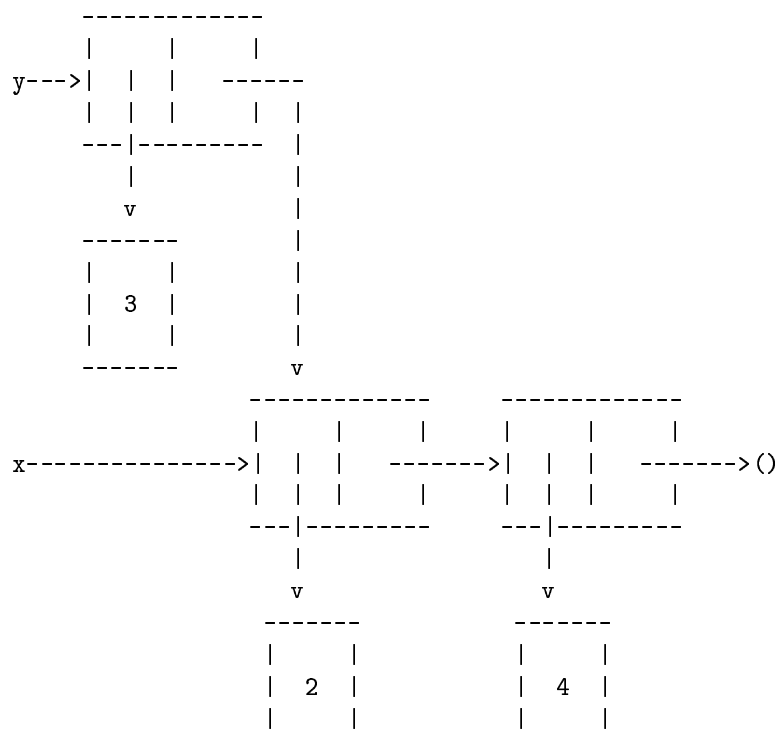


Here the value of `x` is a pointer to the first of two cons cells, each consisting of a car pointer (the left side) and a cdr pointer (the right side.) As in *Concrete Abstractions*, we show a pointer to the null list `()` if the cdr is null. To find the car of the list `x`, we follow the car pointer of the first cons cell to the data value 2. To find the cdr of the list `x`, we follow the cdr pointer of the first cons cell, which points to the second cons cell. To find the next element of the list `x`, we follow the car pointer from the second cons cell, which leads to the data value 4. The cdr pointer of the second cons cell points to the null list, so there are no more elements in the list `x`.

If we now execute

```
(define y (cons 3 x))
```

then the `cons` creates a new cons cell, and sets its car pointer to point to 3 and its cdr pointer to be the value of `x`, resulting in the structure:



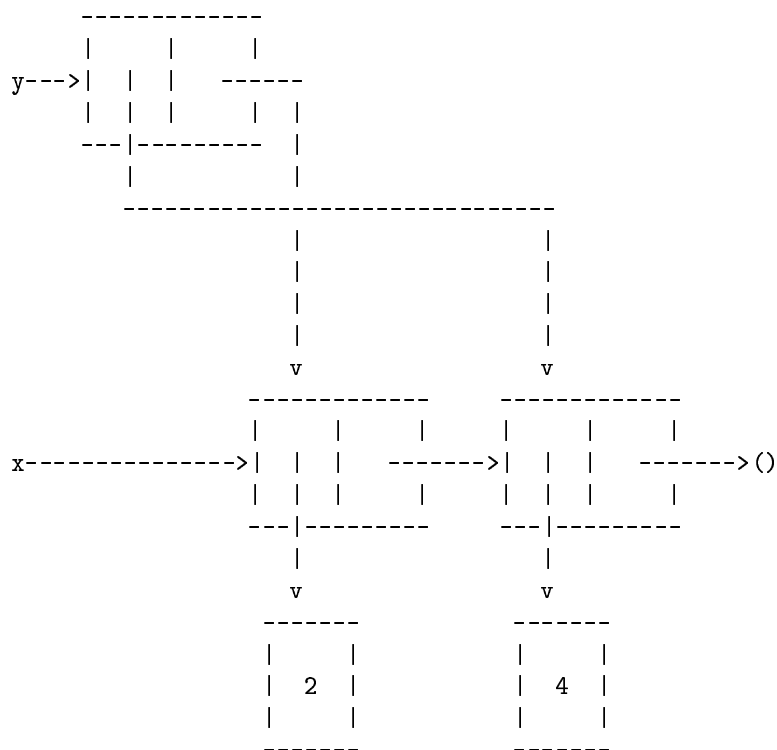
Note that the list that is the value of `x` is a substructure of the list that is the value of `y`. This really only becomes relevant if you can mutate the values in a list, which `set-car!` and `set-cdr!` allow you to do.

The mutators `set-car!` and `set-cdr!`

The mutators `set-car!` and `set-cdr!` change the values of the car and cdr portions of a cons cell, respectively. For the example shown in the preceding section, the result of

```
(set-car! y (cdr x))
```

is to replace the car pointer of `y` with the cdr pointer of `x`, resulting in the structure:



The value of `x` is unchanged, but we now have:

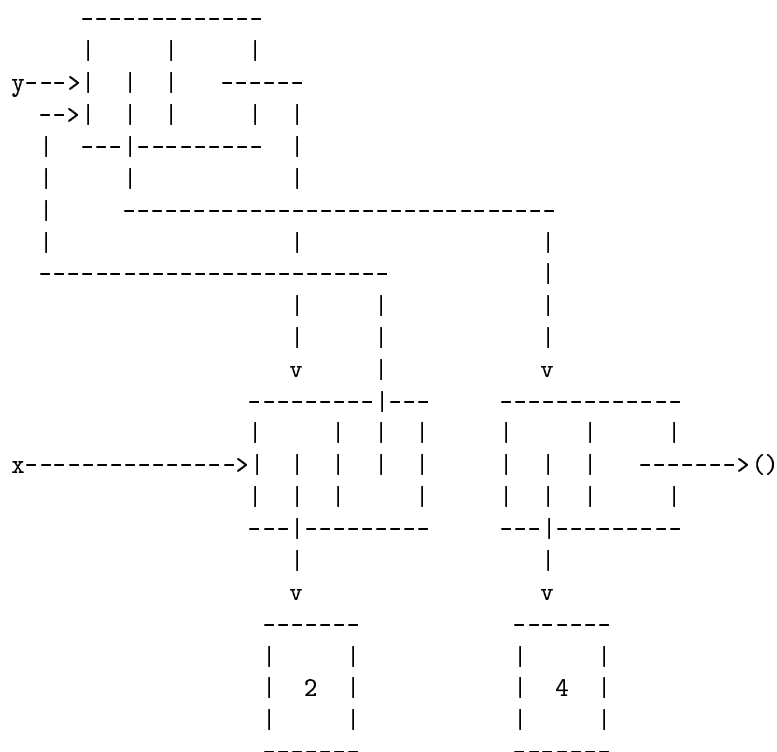
```
y => ((4) 2 4)
```

To see this, note that when we follow the car pointer for `y`, we get a pointer to the list `(4)`, so this is the first element of the list that is the value of `y`. The cdr pointer of `y` points to the list `(2 4)`, so these are the rest of the elements on the list.

If we now execute the command:

```
(set-cdr! x y)
```

we set the cdr pointer of `x` to point to the same place as `y`, which results in the structure:



Note that this structure has loops of pointers in it; nothing prevents us from doing that. However, if we try to evaluate `y` in scm, what is printed is:

```
((4) 2 (4) 2 (4) 2 (4) 2 (4) 2 (4) 2 (4) 2 (4) 2 (4) 2 (4) ...
```

ad infinitum (at which point it is useful to remember control C.)

A queue implementation

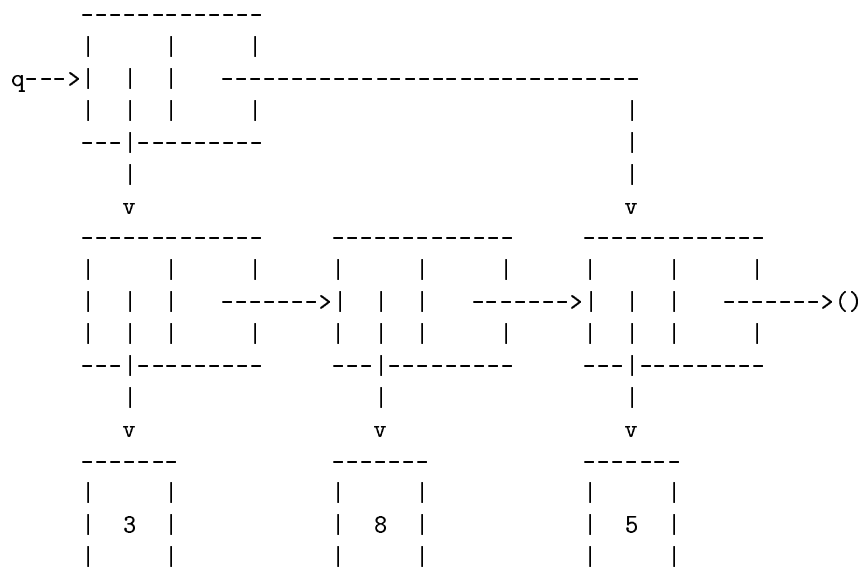
Recall from a previous lecture our implementation of a stack object using a list to represent a stack. The stack operations were implemented with the basic list operations, `car` to access the top of the stack, `cdr` to pop the top element off the stack, and `cons` to add a new top element to the stack. Now we can see that each stack operation runs in time $O(1)$ independent of the number of elements in the stack.

A queue is another abstract data structure whose state is a finite sequence of elements. One end of the finite sequence is the head; the other end is the tail. Elements may be removed only from the head end, and added only at the tail end. The operations test whether the queue is empty, get the value of the head of the queue, remove the head element of the queue (making the next element in sequence the new head element), and add a new element after the tail of the queue. If we choose to represent a queue with the elements

```
3, 8, 5
^      ^
head  tail
```

by the list (3 8 5), then accessing and removing the head element can be done with `car` and `cdr`, each an $O(1)$ operation, but adding a new last element to the list seems to require time proportional to the number of elements in the queue.

To avoid this, we'd like to have "direct access" to the last element of the queue. We'd like a structure that looks as follows, where `q` is the variable naming the queue.



Question: can we create this structure in Scheme without using `set-car!` and `set-cdr!`? Assuming we have created this structure, how can we implement the queue operations? The head of the queue is found by following the `car` pointer from `q` and then the `car` pointer from that cons cell, that is

```
head of q is (car (car q)) => 3
```

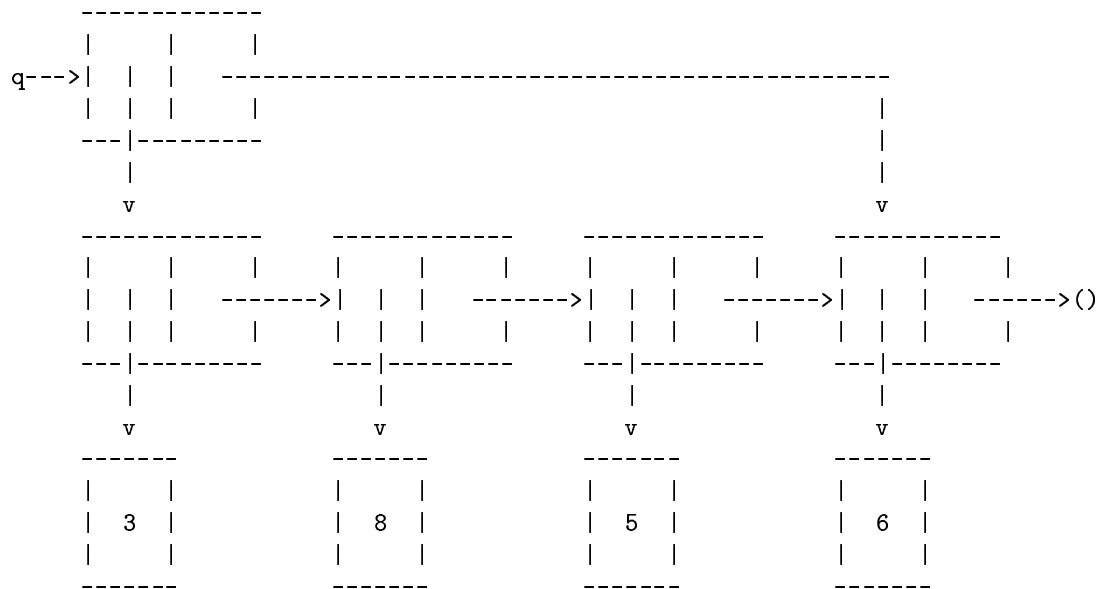
Removing the head of the queue is just setting the `car` pointer of `q` to point to the next cons cell, i.e., `(cadr q)`.

```
dequeue of q is achieved by (set-car! q (cdr (car q)))
```

Adding a new value, say 6, to the end of the queue involves creating a new cons cell whose value is the list (6), setting the `cdr` pointer of the last element of the queue to point to the new cons cell, and finally, making the `cdr` pointer of `q` point to the new cons cell.

```
enqueue 6 on q is achieved by
(let ((new (cons 6 '())))
  (set-cdr! (cdr q) new)
  (set-cdr! q new))
```

The effect of this operation would be as follows.



The operations involve special cases when queue is empty. Nonetheless, all operations can be implemented in time $O(1)$, making this an efficient implementation of a queue.