A problem with vectors

What if we want to add 6 between 5 and 7?

No can do! Elements 7, 8, 9, 10, and 11 are all in the way, and the vector is full.

Need to create a new, bigger vector, and copy everything over…
A problem with vectors

What if we want to add 6 between 5 and 7?
A problem with vectors

What if we want to add 6 between 5 and 7?
No can do! Elements 7, 8, 9, 10, and 11 are all in the way, and the vector is full.
What if we want to add 6 between 5 and 7?
No can do! Elements 7, 8, 9, 10, and 11 are all in the way, and the vector is full.
Need to create a new, bigger vector, and copy everything over…
Books on a string
Books on a string

---

**The Art of Computer Programming**  
*D. E. Knuth*  
*Volume 1: Fundamental Algorithms*  
*Third Edition*  
*Donald E. Knuth*

**SURREAL NUMBERS**  
*Donald E. Knuth*

**The Art of Computer Programming**  
*D. E. Knuth*  
*Volume 2: Seminumerical Algorithms*  
*Third Edition*  
*Donald E. Knuth*

**CONCRETE MATHEMATICS**  
*A Foundation for Computer Science*  
*Donald E. Knuth, Graham, Knuth, Patashnik*  
*Second Edition*
Books on a string

The Art of Computer Programming
D.E. Knuth

SURREAL NUMBERS
D.E. KNUTH

The Art of Computer Programming
Volume 2
Seminumerical Algorithms
Third Edition
Donald E. Knuth

CONCRETE MATHEMATICS
A Foundation for Computer Science
Graham Knuth Patashnik
Second Edition
Nodes and pointers

You saw cons in 111.

- **car** holds the *first* element, and
- **cdr** holds a pointer to the *rest* of the list.

![Diagram showing nodes and pointers with elements 2, 4, 6, and 8]
Nodes and pointers

Inserting in the middle? No problem!
Just change the pointers

```
2
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
</table>
car

cdr

4
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
</table>
car

cdr

6
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
</table>
car

cdr

8
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
</table>
car

cdr

5
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
</table>
car

cdr
```

- car holds the first element, and
- cdr holds a pointer to the rest of the list.
Nodes and pointers

Inserting in the middle? No problem!
Just change the pointers

- car holds the first element,
- cdr holds a pointer to the rest of the list.
Nodes and pointers

Inserting in the middle? No problem!
Just change the pointers
Nodes and pointers

Inserting in the middle? No problem!
Just change the pointers

---

You saw `cons` in 111.

- `car` holds the first element,
- `cdr` holds a pointer to the rest of the list.

Inserting in the middle? No problem!
Just change the pointers

```
<table>
<thead>
<tr>
<th>car</th>
<th>cdr</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>
```
Nodes and pointers

Inserting in the middle? No problem!
Just change the pointers

[Diagram showing nodes and pointers with arrows connecting them, illustrating the concept of inserting in the middle.]
Inserting at the beginning

```lisp
(cons 8 (cons 7 (cons 6 (cons 5 (cons 4 (cons 3 (cons 2 '() '()))))))
```
Inserting at the beginning
Inserting at the beginning
Inserting at the beginning
Inserting at the beginning
Indirection

head

lst

1 2 3 4 5 6 7 8
Indirection
Now in DSSL2
Linked lists in DSSL2

# Link is one of:
# - node { data: Number, next: Link }
# - None

struct node:
    let data
    let next

class SLL:
    let head

    def __init__(self):
        self.head = None
Linked lists in DSSL2

# Link is one of:
# - node { data: Number, next: Link }
# - None

struct node:
    let data
    let next

class SLL:
    let head

    def __init__(self):
        self.head = None

    def push_front(self, data):
        self.head = node(data, self.head)
class SLL:
    ...

    def get_front(self):
        if node?(self.head): self.head.data
        else: error('SLL.get_front: empty list')

    def get_nth(self, n):
        curr = self.head
        while n > 0:
            if curr is None:
                error('SLL.get_nth: too short')
            curr = curr.next
            n = n - 1
        curr.data
class SLL:
    ...

    def get_front(self):
        if node?(self.head): self.head.data
        else: error('SLL.get_front: empty list')

    def get_nth(self, n):
        curr = self.head
        while n > 0:
            if curr is None:
                error('SLL.get_nth: too short')
            curr = curr.next
            n = n - 1
        curr.data
More DSSL2 list operations

A (re)factoring:

class SLL:

    ...

    def _find_nth_node(self, n):
        curr = self.head
        while n > 0:
            if curr is None: error('too short')
            curr = curr.next
            n = n - 1
        curr

    def get_nth(self, n):
        self._find_nth_node(n).data

    def set_nth(self, n, val):
        self._find_nth_node(n).data = val
What else might we want to do?
What else might we want to do?

- Insert or remove at the given position or the end.
- Split a list in two or splice two into one.
- Know how long the list is without counting.
Keeping the length

How can we make sure the len field is always right?
Keeping the length

How can we make sure the \texttt{len} field is always right?
Quick access to the tail

Which operations are simple now? Which are still more work?
Quick access to the tail

head

tail

len

6

Which operations are simple now? Which are still more work?
Doubly-linked
Circular, doubly-linked with sentinel

len 6

data prev next
1 2 3 4 5 6

sentinel

prev next data

Empty (circular, doubly-linked w/sentinel)
Let’s look at a singly-linked list class in DSSL2.
Next time: abstract data types