List Zippers

Dr. Mattox Beckman

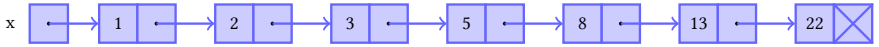
Illinois Institute of Technology
Department of Computer Science

Objectives

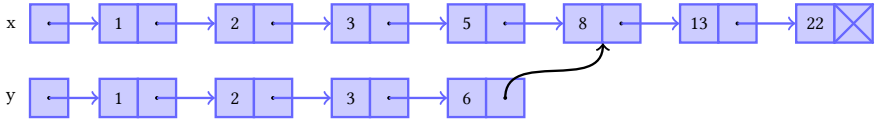
- Describe why immutable doubly linked lists are hard.
- Describe how zippers simulate forward and backward motion.
- Describe how zippers simulate mutation.

Modifying a Data Structure

- Consider this immutable list.



- If we want to update the 5 to be 6, we must copy everything to the root.



- The rule is: if I can reach the modified node from here, it needs to be rebuilt.

What about doubly linked lists?

- What about this list?

- Updating 2 to 5, both directions need to be copied!

A useful programming trick.

- "How can I update a doubly linked list?" is a bad question.
  - Too specific!
- "How can I move backwards and forwards through some data?" is better.
  - Not tied to an implementation.
- So.. how do I do it?

Hansel and Gretel

- Suppose we are "at" the 3 node.
- Going forward, we could see 5 and 9.
- Going backwards, we could see 2 and 1.
  - We would see them in the reverse order...

Enter the Zipper

```
(defrecord ListZip [before after])
(defn make-list-zip [x]
   (ListZip. '() x))
(defn current [z] (first (:after z)))
(defn forward [z]
   (ListZip. (cons (-> z :after first) (:before z))
             (rest (:after z))))
(defn backward [x]
   (ListZip. (rest (:after z))
             (cons (-> z :before first) (:after z))))
```


Sample Run

```
(def x (make-list-zip '(1 2 3 4 5)))
;; => #'user/x
x
;; => #user.ListZip{:before (), :after (1 2 3 4 5)}
(current x)
;; => 1
(forward x)
;; => #user.ListZip{:before (1), :after (2 3 4 5)}
(-> x forward forward)
;; => #user.ListZip{:before (2 1), :after (3 4 5)}
```

Updating

- Update is in $\mathcal{O}(1)$ time!

```
(defn update [z elt]
      (ListZip. (:before z) (cons elt (-> z :after rest))))
(def x2 (-> x forward forward))
;; => #'user/x2
(update x2 20)
;; => #user.ListZip{:before (2 1), :after (20 4 5)}
(forward (update x2 20))
;; => #user.ListZip{:before (20 2 1), :after (4 5)}
```


Back to lists

```
(defn list-zip-to-list [z]
      (concat (reverse (:before z)) (:after z)))

(list-zip-to-list  (forward (update x2 20)))
;; => (1 2 20 4 5)
```


Your turn!

- Type in this code and play with it!
- For in-class activity: using zippers, write a search function that takes a list and an element and returns a list with seven items: the three elements before the one you found, the one you found, and the three elements after the one you found.

```
(zip-search '(2 4 6 8 10 12 14 16 18 20) 12)
;; => '(6 8 10 12 14 16 18)
```