Immutable Linked Lists

Dr. Mattox Beckman

Illinois Institute of Technology
Department of Computer Science

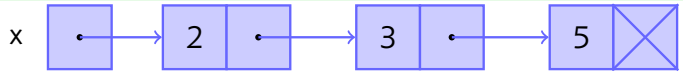
Objectives

- Learn some of the basic list operations for immutable lists.
  - Creation
  - "Update"
  - Selection
- Show how memory is allocated when multiple lists are combined.
- Understand the tradeoffs inherint in immutable lists.

In the beginning

- An empty list is represented by `nil`.
- A list with data is created by the `cons` function.
  - "A list is an item with a pointer to another list."
- Clojure uses `cons` to create sequences, so we will use our own record instead. Don't confuse `cons` with `Cons.` !

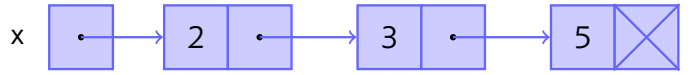
```
1 (defrecord Cons [first rest])
2 (def x (Cons. 2 (Cons. 3 (Cons. 5 nil))))
3 ;; => #user.Cons{:first 2,
4 ;;                :rest #user.Cons{:first 3,
5 ;;                                 :rest #user.Cons{:first
```



Inserting

- Inserting can only be done at the beginning of the list.
- And even then, we're not really inserting...

```
1 (def x (Cons. 2 (Cons. 3 (Cons. 5 nil))))
2 (def y (Cons. 8 x))
```



- We are not modifying the original.
- But we are sharing the data!

Standard Function: nth

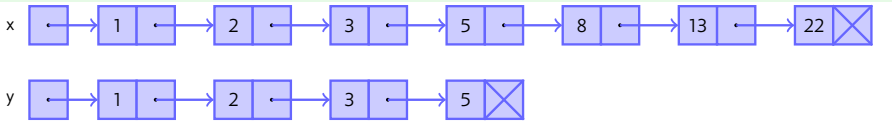
- Lists are not arrays, but sometimes you want to pretend.

```
(defn my-nth [n xx]
  (if (= n 0) (:first xx)
      (my-nth (- n 1) (:rest xx))))
(my-nth 1 x)
;; => 1
(nth 3 '(1 2 3 5 6 8 3))  ; built-in version
;; => 5
```


Standard Function: take

- It is common to want the first *n* elements of a list.
- Can you write `mklist`, as shown below?

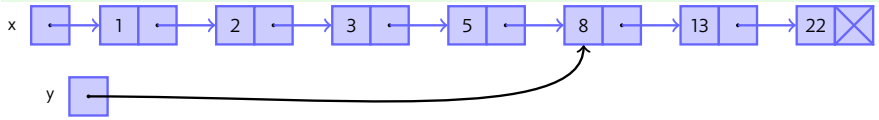
```
(defn my-take [xx n]
  (if (= n 0) nil
     (Cons. (:first xx) (my-take (:rest xx) (- n 1)))))
(def x (mklist 1 2 3 5 8 13 22))
(def y (my-take x 4)) ; built-in take, uses sequences
```



Standard Function: drop

- It is common to want everything *but* the first *n* elements of a list.

```
(defn my-drop [xx n]
  (if (= n 0) xx
     (my-drop (:rest xx) (- n 1))))
(def x (mklist 1 2 3 5 8 13 22))
(def y (my-drop x 4))
```

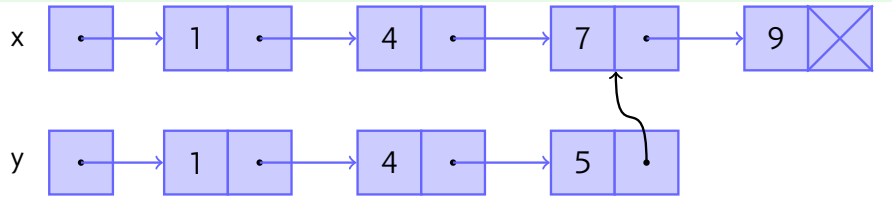


- Note that *y* shares elements with *x*.

Inserting in the middle

- What if we try to insert in the middle anyway?

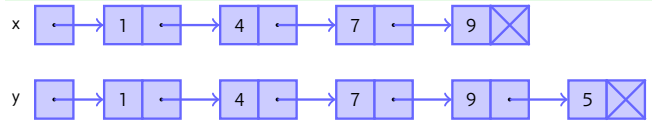
```
(defn insert [xx elt]
    (cond (empty? xx) (list elt)
          (< elt (:first xx)) (Cons. elt xx)
          :else (Cons. (:first xx)
                       (insert (:rest xx) elt))))
(def x (mklist 1 4 7 9))
(def y (insert x 5))
```



Inserting at the end

- Inserting at the end is pretty bad...

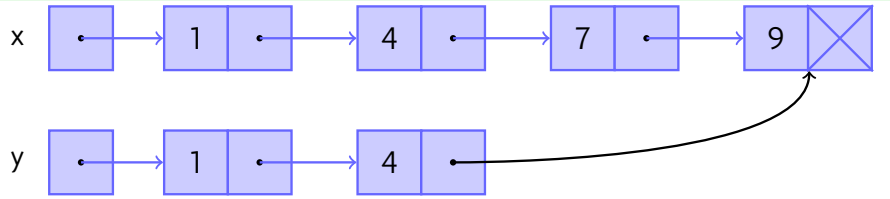
```
(defn insert-at-end [xx elt]
   (cond (nil? xx) (list elt) ; use empty? for sequences!!
         :else (Cons. (:first xx)
                      (insert-at-end (:rest xx) elt))))
(def x (mklist 1 4 7 9))
(def y (insert-at-end x 5))
```



Deletion

- Here is an example of delete.

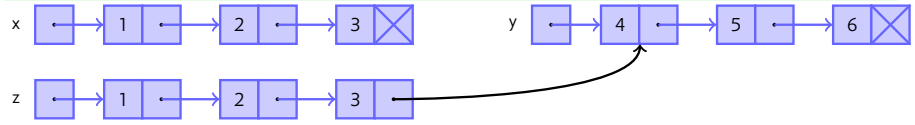
```
(defn delete [xx elt]
   (cond (nil? xx) nil ; use empty? for sequences!
         (= elt (:first xx)) (:rest xx)
         :else (Cons. (:first xx) (delete (:rest xx) elt))))
(def x (mklist 1 4 7 9))
(def y (delete x 7))
```



Appending

- Append is very much like insert-at-end.
- Avoid it if you can!

```
(defn append [xx yy]
   (if (nil? xx) yy
       (Cons. (:first xx) (append (:rest xx) yy))))
(def x (mklist 1 2 3))
(def y (mklist 4 5 6))
(def z (append x y))
```



Reverse

```
(defn bad-reverse [xx]
   (if (nil? xx) nil
      (append (bad-reverse (:rest xx)) (Cons. (:first xx)
```

- This is the näive way of reversing a list.
- It runs in $\mathcal{O}(n^2)$ time though!
  - Append is $\mathcal{O}(n)$ in the size of the first list.
  - ... and append is called once for every element in the list.

Reversing the right way

```clojure
(defn reverse "Reverse a list."
   ([xx]
      (reverse xx nil))
   ([xx acc]
      (if (nil? xx) acc ; use empty? for sequences!
          (reverse (:rest xx) (Cons. (:first xx) acc)))))

(reverse (mklist 1 2 3))
```