

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 inherent in immutable lists.

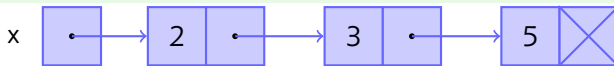
In the beginning

- ▶ 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.

```
1 (defn my-nth [n xx]
2   (if (= n 0) (:first xx)
3     (my-nth (- n 1) (:rest xx))))
4 (my-nth 1 x)
5 ;; => 1
6 (nth 3 '(1 2 3 5 6 8 3)) ; built-in version
7 ;; => 5
```

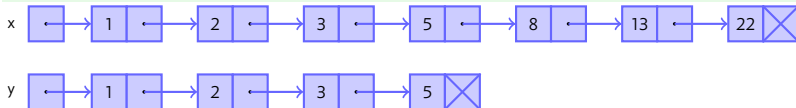
Standard Function: take

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

```

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

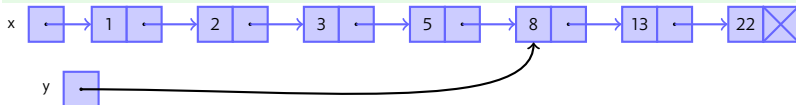
```



Standard Function: drop

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

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



- Note that y shares elements with x .

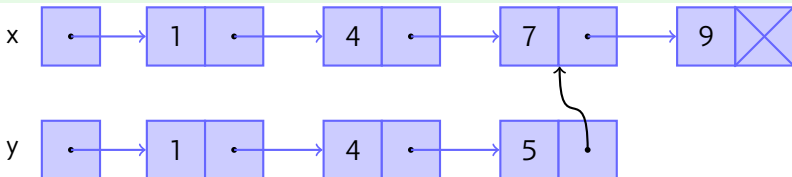
Inserting in the middle

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

```

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

```



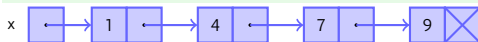
Inserting at the end

- Inserting at the end is pretty bad...

```

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

```



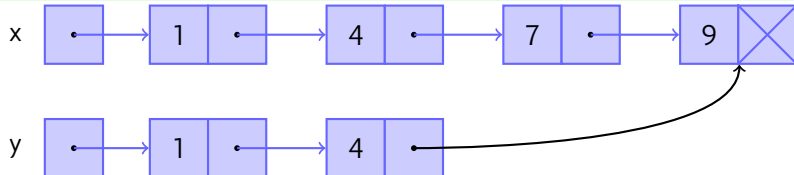
Deletion

- Here is an example of delete.

```

1 (defn delete [xx elt]
2   (cond (nil? xx) nil ; use empty? for sequences!
3         (= elt (:first xx)) (:rest xx)
4         :else (Cons. (:first xx) (delete (:rest xx) elt))
5   )
6 (def x (mklist 1 4 7 9))
7 (def y (delete x 7))

```



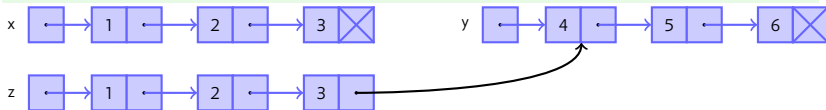
Appending

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

```

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

```



Reverse

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

- ▶ This is the naïve 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

```

1 (defn reverse "Reverse a list."
2   ([xx]
3     (reverse xx nil))
4   ([xx acc]
5     (if (nil? xx) acc ; use empty? for sequences!
6         (reverse (:rest xx) (Cons. (:first xx) acc))))))
7
8 (reverse (mklist 1 2 3))

```

