Doubly Linked Lists

Dr. Mattox Beckman

Illinois Institute of Technology
Department of Computer Science

Objectives

- Understand how to create a doubly linked list.
- Be able to write insertion code.
- Be able to write deletion code.
- Be able to express the tradeoff between doubly linked lists and singly linked lists.

Doubly Linked Lists

- Conceptually not much different than singly linked lists.
- They have two pointers: previous and next.
- Always mutable!

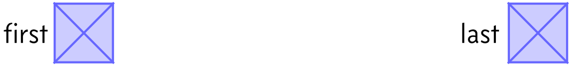


Figure : A boring empty list.

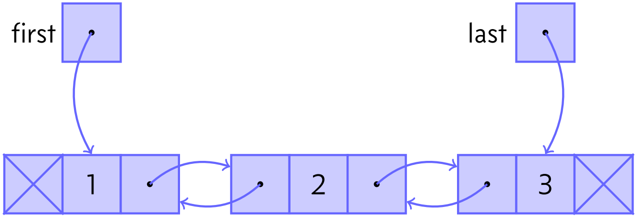


Figure : Elements 1,2, and 3.

Building the ADT

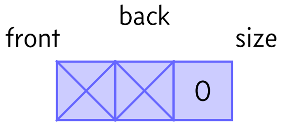
- We should keep track of front, back, and size.
- Doubly linked lists should be mutable.

```
(deftype DList [^{:unsynchronized-mutable true} front
                ^{:unsynchronized-mutable true} back
                ^{:unsynchronized-mutable true} size]
   DListP ;; accessors here...
)
(deftype DNode [^{:unsynchronized-mutable true} prev
                ^{:unsynchronized-mutable true} data
                ^{:unsynchronized-mutable true} next]
   DNodeP ;; accessors here...
)
(defn make-dlist []
   (DList. nil nil 0))
(defn make-dnode [prev data next]
   (DNode. prev data next))

```

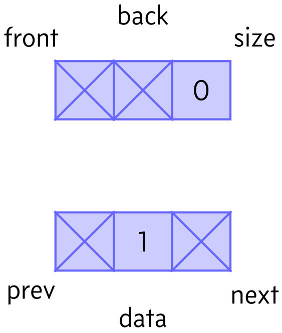
Adding

- There are two cases to adding. For empty list:
  1. Create the node.
  2. Set front to point to node.
  3. Set back to point to node.
  4. Set size to one.



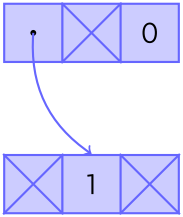
Adding

- There are two cases to adding. For empty list:
  1. Create the node.
  2. Set front to point to node.
  3. Set back to point to node.
  4. Set size to one.



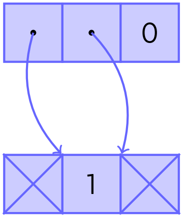
Adding

- There are two cases to adding. For empty list:
    1. Create the node.
    2. Set front to point to node.
    3. Set back to point to node.
    4. Set size to one.



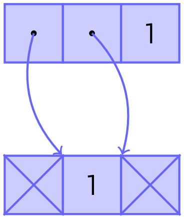
Adding

- There are two cases to adding. For empty list:
  1. Create the node.
  2. Set front to point to node.
  3. Set back to point to node.
  4. Set size to one.



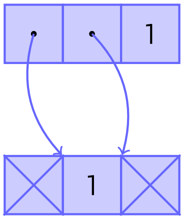
Adding

- There are two cases to adding. For empty list:
  1. Create the node.
  2. Set front to point to node.
  3. Set back to point to node.
  4. Set size to one.



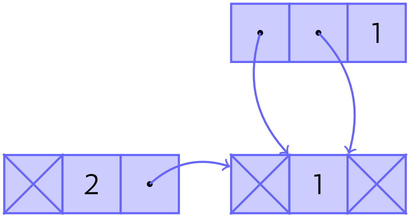
Second Case

- For list with data:
  1. Create the node.
  2. Set next of node to front of list.
  3. Set prev of front node to node.
  4. Set front of list to node.
  5. Increment size.



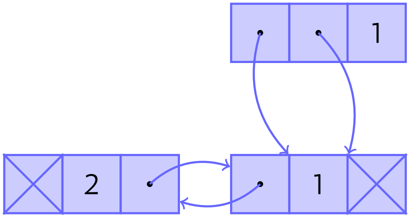
Second Case

- For list with data:
  1. Create the node.
  2. Set next of node to front of list.
  3. Set prev of front node to node.
  4. Set front of list to node.
  5. Increment size.



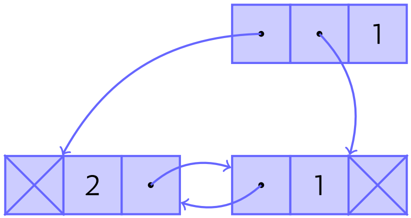
Second Case

- For list with data:
  1. Create the node.
  2. Set next of node to front of list.
  3. Set prev of front node to node.
  4. Set front of list to node.
  5. Increment size.



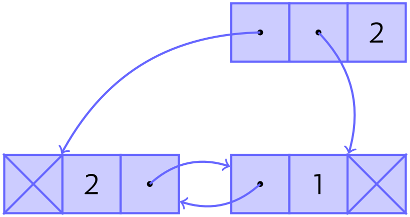
Second Case

- For list with data:
  1. Create the node.
  2. Set next of node to front of list.
  3. Set prev of front node to node.
  4. Set front of list to node.
  5. Increment size.



Second Case

- For list with data:
  1. Create the node.
  2. Set next of node to front of list.
  3. Set prev of front node to node.
  4. Set front of list to node.
  5. Increment size.



Adding

- Here is the code for adding to the front.

```
(defn insert-front [dlist elt]
  (let [node (make-dnode nil elt (front dlist))]
    (if (nil? (front dlist))
        (do (set-front! dlist node)
            (set-back!  dlist node)
            (set-size! dlist (+ 1 (size dlist))))
        (do (set-prev! (front dlist) node)
            (set-front! dlist node)
            (set-size! dlist (+ 1 (size dlist)))))))
```

Sample Run

```
(def xx (make-dlist))
;; => #'user/xx
(insert-front xx 10)
;; => 1
(identical? (-> xx front) (-> xx back) )
;; => true
(insert-front xx 20)
;; => 3
(identical? (-> xx :front) (-> xx :back) )
;; => false
(-> xx front data)
;; => 20
(-> xx front next data)
;; => 10
```

Find

- We can search from the front or from the back.

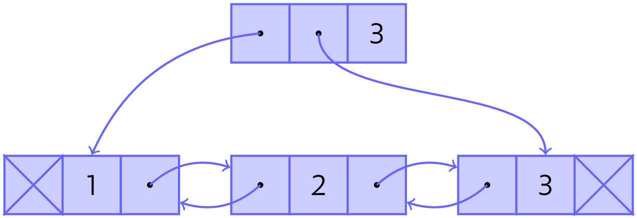
```
(defn find-fwd [dnode elt]
  (cond (nil? dnode)             false
        (= (data dnode) elt)     true
        :fine-be-that-way        (find-fwd (next dnode) elt))
```

Deletion

- There are three edge cases for delete!
  - Delete beginning
  - Delete end
  - Delete only
- Important because you have to do different things on the edge than in the middle.
- Sentinels will rescue us later.

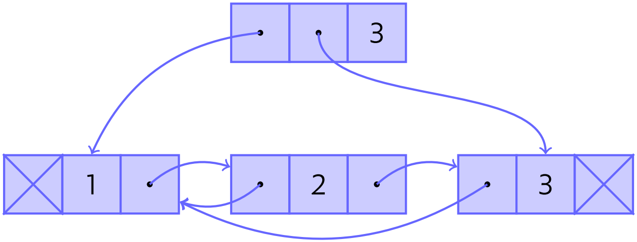
Example: delete 2

- Set next's prev to prev
- Set prev's next to next
- Decrement size



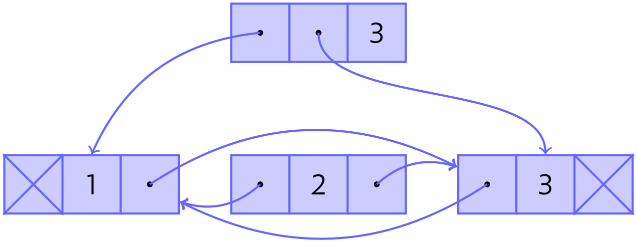
Example: delete 2

- Set next's prev to prev
- Set prev's next to next
- Decrement size



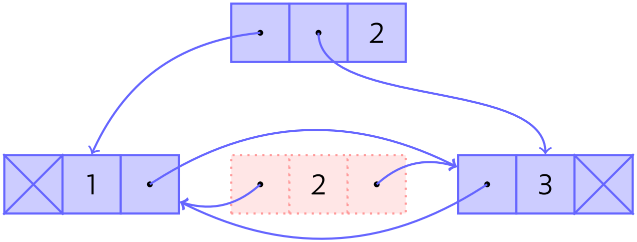
Example: delete 2

- Set next's prev to prev
- Set prev's next to next
- Decrement size



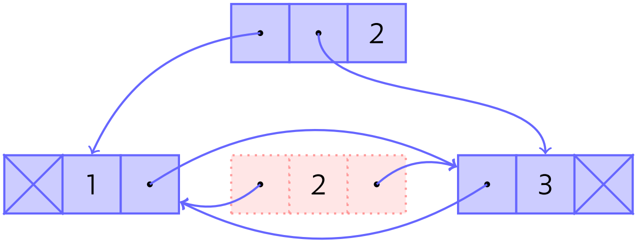
Example: delete 2

- Set next's prev to prev
- Set prev's next to next
- Decrement size



Example: delete 2

- Set next's prev to prev
- Set prev's next to next
- Decrement size



The 2 node is garbage now.