

Doubly-Linked Lists



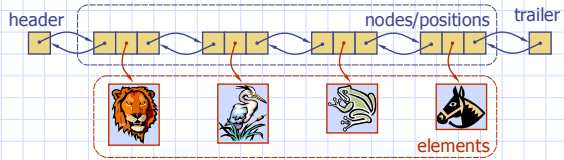
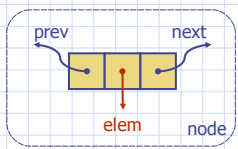
© 2013 Goodrich, Tamassia, Goldwasser

Doubly-Linked Lists

1

Doubly Linked List

- A doubly linked list provides a natural implementation of the Node List ADT
- Nodes implement Position and store:
 - element
 - link to the previous node
 - link to the next node
- Special trailer and header nodes



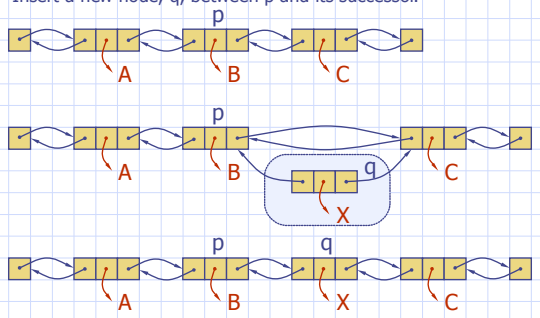
© 2013 Goodrich, Tamassia, Goldwasser

Doubly-Linked Lists

2

Insertion

- Insert a new node, q, between p and its successor.



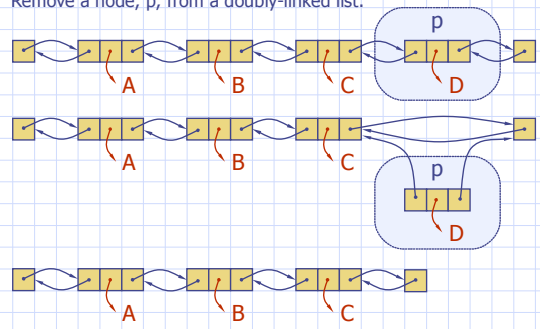
© 2013 Goodrich, Tamassia, Goldwasser

Doubly-Linked Lists

3

Deletion

- Remove a node, p, from a doubly-linked list.



© 2013 Goodrich, Tamassia, Goldwasser

Doubly-Linked Lists

4

Doubly-Linked List in Python

```

1 class _DoublyLinkedBase:
2     """A base class providing a doubly linked list representation."""
3
4     class _Node:
5         """Lightweight, nonpublic class for storing a doubly linked node."""
6         (omitted here; see previous code fragment)
7
8     def __init__(self):
9         """Create an empty list."""
10        self._header = self._Node(None, None, None)
11        self._trailer = self._Node(None, None, None)
12        self._header._next = self._trailer # trailer is after header
13        self._trailer._prev = self._header # header is before trailer
14        self._size = 0 # number of elements
15
16    def __len__(self):
17        """Return the number of elements in the list."""
18        return self._size
19
20    def is_empty(self):
21        """Return True if list is empty."""
22        return self._size == 0
23
24    def _insert_between(self, e, predecessor, successor):
25        """Add element e between two existing nodes and return new node."""
26        newest = self._Node(e, predecessor, successor) # linked to neighbors
27        predecessor._next = newest
28        successor._prev = newest
29        self._size += 1
30        return newest
31
32    def _delete_node(self, node):
33        """Delete nonsentinel node from the list and return its element."""
34        predecessor = node._prev
35        successor = node._next
36        predecessor._next = successor
37        successor._prev = predecessor
38        self._size -= 1
39        element = node._element # record deleted element
40        node._prev = node._next = node._element = None # deprecate node
41        return element # return deleted element

```

Performance

- In a doubly linked list
 - The space used by a list with n elements is $O(n)$
 - The space used by each position of the list is $O(1)$
 - All the standard operations of a list run in $O(1)$ time

Positional List

- To provide for a general abstraction of a sequence of elements with the ability to identify the location of an element, we define a **positional list** ADT.
- A position acts as a marker or token within the broader positional list.
- A position p is unaffected by changes elsewhere in a list; the only way in which a position becomes invalid is if an explicit command is issued to delete it.
- A position instance is a simple object, supporting only the following method:
 - `p.element()`: Return the element stored at position p .

Positional Accessor Operations

- `L.first()`: Return the position of the first element of L , or `None` if L is empty.
- `L.last()`: Return the position of the last element of L , or `None` if L is empty.
- `L.before(p)`: Return the position of L immediately before position p , or `None` if p is the first position.
- `L.after(p)`: Return the position of L immediately after position p , or `None` if p is the last position.
- `L.is_empty()`: Return `True` if list L does not contain any elements.
- `len(L)`: Return the number of elements in the list.
- `iter(L)`: Return a forward iterator for the *elements* of the list. See Section 1.8 for discussion of iterators in Python.

Positional Update Operations

- `L.add_first(e)`: Insert a new element `e` at the front of `L`, returning the position of the new element.
- `L.add_last(e)`: Insert a new element `e` at the back of `L`, returning the position of the new element.
- `L.add_before(p, e)`: Insert a new element `e` just before position `p` in `L`, returning the position of the new element.
- `L.add_after(p, e)`: Insert a new element `e` just after position `p` in `L`, returning the position of the new element.
- `L.replace(p, e)`: Replace the element at position `p` with element `e`, returning the element formerly at position `p`.
- `L.delete(p)`: Remove and return the element at position `p` in `L`, invalidating the position.

Positional List in Python

```
1 class PositionalList, DoublyLinkedListBase:
2     """A sequential container of elements allowing positional access."""
3
4     #----- nested Position class -----
5     class Position:
6         """An abstraction representing the location of a single element."""
7
8         def __init__(self, container, node):
9             """Constructor should not be invoked by user."""
10            self._container = container
11            self._node = node
12
13        def element(self):
14            """Return the element stored at this Position."""
15            return self._node._element
16
17        def __eq__(self, other):
18            """Return True if other is a Position representing the same location."""
19            return type(other) is type(self) and other._node is self._node
20
21        def __ne__(self, other):
22            """Return True if other does not represent the same location."""
23            return not (self == other) # opposite of __eq__
24
25        #----- utility method -----
26        def validate(self, p):
27            """Return position's node, or raise appropriate error if invalid."""
28            if not isinstance(p, self.Position):
29                raise TypeError('p must be proper Position type')
30            if p._container is not self:
31                raise ValueError('p does not belong to this container')
32            if p._node._next is None: # convention for deprecated nodes
33                raise ValueError('p is no longer valid')
34            return p._node
```

Positional List in Python, Part 2

```
35 #----- utility method -----
36 def _make_position(self, node):
37     """Return Position instance for given node (or None if sentinel)."""
38     if node is self._header or node is self._trailer: # boundary violation
39         return None
40     else:
41         return self.Position(self, node) # legitimate position
42
43 #----- accessors -----
44 def first(self):
45     """Return the first Position in the list (or None if list is empty)."""
46     return self._make_position(self._header._next)
47
48 def last(self):
49     """Return the last Position in the list (or None if list is empty)."""
50     return self._make_position(self._trailer._prev)
51
52 def before(self, p):
53     """Return the Position just before Position p (or None if p is first)."""
54     node = self._validate(p)
55     return self._make_position(node._prev)
56
57 def after(self, p):
58     """Return the Position just after Position p (or None if p is last)."""
59     node = self._validate(p)
60     return self._make_position(node._next)
61
62 def __iter__(self):
63     """Generate a forward iteration of the elements of the list."""
64     cursor = self.first()
65     while cursor is not None:
66         yield cursor.element()
67         cursor = self.after(cursor)
```

Positional List in Python, Part 3

```
68 #----- mutators -----
69 # override inherited version to return Position, rather than Node
70 def _insert_between(self, e, predecessor, successor):
71     """Add element between existing nodes and return new Position"""
72     node = super()._insert_between(e, predecessor, successor)
73     return self._make_position(node)
74
75 def add_first(self, e):
76     """Insert element e at the front of the list and return new Position."""
77     return self._insert_between(e, self._header, self._header._next)
78
79 def add_last(self, e):
80     """Insert element e at the back of the list and return new Position."""
81     return self._insert_between(e, self._trailer._prev, self._trailer)
82
83 def add_before(self, p, e):
84     """Insert element e into list before Position p and return new Position."""
85     original = self._validate(p)
86     return self._insert_between(e, original._prev, original)
87
88 def add_after(self, p, e):
89     """Insert element e into list after Position p and return new Position."""
90     original = self._validate(p)
91     return self._insert_between(e, original, original._next)
92
93 def delete(self, p):
94     """Remove and return the element at Position p"""
95     original = self._validate(p)
96     return self._delete_node(original) # inherited method returns element
97
98 def replace(self, p, e):
99     """Replace the element at Position p with e.
100    Return the element formerly at Position p.
101
102    original = self._validate(p)
103    old_value = original._element # temporarily store old element
104    original._element = e # replace with new element
105    return old_value # return the old element value
106
```