05 Singly Linked Lists

Instructor : Ke Wei（柯韋）

A319 ✆ Ext. 6452 ✉ wke@ipm.edu.mo

http://brouwer.ipm.edu.mo/COMP122/19/

Bachelor of Science in Computing, School of Public Administration, Macao Polytechnic Institute

January 21, 2019

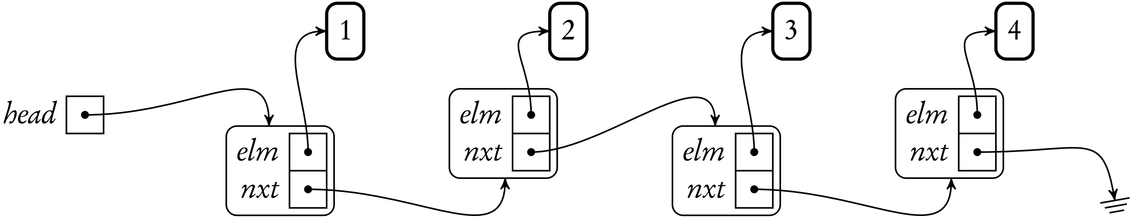
Outline

Mutable Collections

- Many data structures deal with collections of data, to group relevant items together.
- While iteration is the main operation of immutable collections, there are more operations for mutable collections.
- Items can be *added*, *removed* and *retrieved* from a mutable collection.
- In practice, data items must be organized in a structure, so that the above operations can be performed in a specific way by computer programs.
- The simplest way to organize items is to put them one after another, such as in a list.

Singly Linked Lists

- An array allocates memory for all its elements put together as one block of memory.
- In contrast, a linked list allocates space for each element separately in its own instance memory called a *node*.
- The list gets its overall structure by using object *references* to connect all its nodes together like the *links* in a chain.
- In a singly linked list, each node has only one link which points to the next node, and the link in the last node contains `None`, which is usually illustrated as $\perp$.



Elements and Nodes

- A pure item (payload) in a collection (here, a linked list) is called an element.
- There are also *helpers* to maintain the structure of the collection, for example, the links.
- An element and its associated helper forms a node.
- From the abstraction point of view, elements can be seen from outside, yet the structure of a node is internal.

Nodes in Singly Linked Lists

- The node is defined as a class *Node*, the element of the node can be anything, for example, an integer or a string.
- The *nxt* field is a reference to an instance of class *Node* itself. Hence, *Node* is a *recursive data type*.

```python
class Node:
    def __init__(self, elm, nxt):
        self.elm = elm
        self.nxt = nxt
```

The *LnLs* Class

- We use a field *head* to point to the first node of a linked list.
- The *head* is initially `None`, representing an empty list.
- If the list is not empty, we can access the first element immediately.
- The first element is often called the *top* element.

```
class LnLs:
    def __init__(self):
        self.head = None

    def __bool__(self):
        return self.head is not None
```

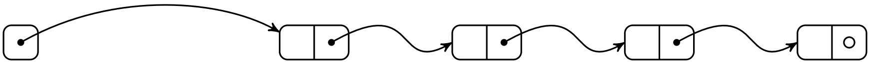
```
    def top(self):
        if not self:
            raise IndexError
        return self.head.elm
```

Insertion and Deletion

- The most efficient insertion and deletion of a singly linked list happen at the head position.
- To insert an element to the head position is called a *push*.
- To delete an element from the head position is called a *pop*.

```
def push(self, x):
    p = Node(x, self.head)
    self.head = p
```

```
def pop(self):
    x = self.top()
    self.head = self.head.nxt
    return x
```

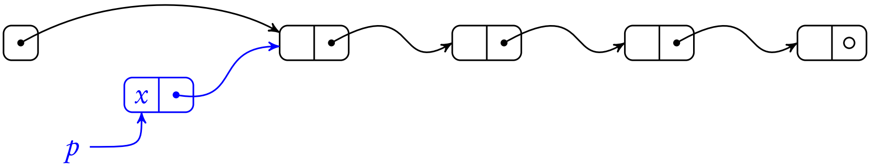


Insertion and Deletion

- The most efficient insertion and deletion of a singly linked list happen at the head position.
- To insert an element to the head position is called a *push*.
- To delete an element from the head position is called a *pop*.

```python
def push(self, x):
    p = Node(x, self.head)
    self.head = p
```

```python
def pop(self):
    x = self.top()
    self.head = self.head.nxt
    return x
```

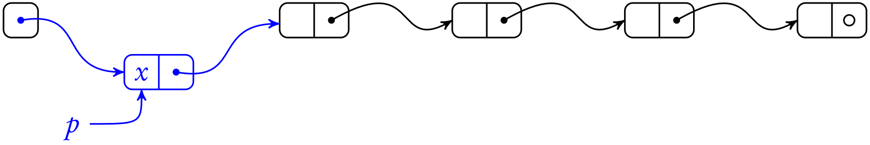


Insertion and Deletion

- The most efficient insertion and deletion of a singly linked list happen at the head position.
- To insert an element to the head position is called a *push*.
- To delete an element from the head position is called a *pop*.

```
def push(self, x):
    p = Node(x, self.head)
    self.head = p
```

```
def pop(self):
    x = self.top()
    self.head = self.head.nxt
    return x
```

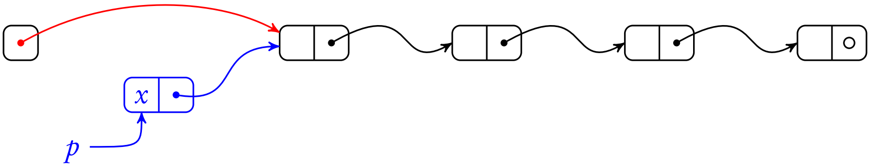


Insertion and Deletion

- The most efficient insertion and deletion of a singly linked list happen at the head position.
- To insert an element to the head position is called a *push*.
- To delete an element from the head position is called a *pop*.

```
def push(self, x):
    p = Node(x, self.head)
    self.head = p
```

```
def pop(self):
    x = self.top()
    self.head = self.head.nxt
    return x
```



Making Linked Lists as Iterables

- Sometimes we want to iterate all the elements in a linked list.
- We don't want to expose the nodes to the outside, otherwise, the nodes can be *tampered*, breaking the structure.
- We yield the elements rather than the nodes.

```
def __iter__(self):
    p = self.head
    while p:
        yield p.elm
        p = p.nxt
```

```
def __init__(self, s = None):
    self.head = None
    if s:
        for x in s:
            self.push(x)
        self.reverse()   # to be defined
```

- We can also do the reverse, construct a linked list from an iterable.

Finding an Element

- One of the most common operations on a collection is to test if the collection contains a certain element.
- We *traverse* the linked list to find the first element that equals to the given one.
- We return the index of the element in the linked list if one is found, otherwise, we return -1.

```
def index_of(self, x):
    for i, y in enumerate(self):
        if x == y:
            return i
    return -1
```

✍ Try to write a method *last_index_of* to return the index of the last element found.

Reversing a List

- If we pop elements from a list and push them to another list, one by one, we get a list with elements in the reversed order. (Left Fig.)

```
def reverse(self):
    rev = LnLs()
    while self:
        rev.push(self.pop())
    self.head = rev.head
```

```
def reverse(self):
    p, q = self.head, None
    # q points to the previous node
    while p is not None:
        t = p.nxt
        p.nxt = q
        q = p
        p = t
    self.head = q
```

- We can also reverse the nodes in-place, to avoid creating new nodes. (Right Fig.)

Using Singly Linked Lists

- Now, we can create a singly linked list and perform some operations on it.

```
>>> ll = LnLs()
>>> ll.push('apple')
>>> ll.push('orange')
>>> ll.push('peach')
>>> list(ll)
['peach', 'orange', 'apple']
>>> ll.find_first('orange')
'orange'
>>> repr(ll.find_first('banana'))
'None'
```

```
>>> ll.pop()
'peach'
>>> ll.push('banana')
>>> list(ll)
['banana', 'orange', 'apple']
>>> ll.reverse()
>>> [x+'*' for x in ll]
['apple*', 'orange*', 'banana*']
```

- We also try to create a linked list from an iterable.

```
>>> list(LnLs(x for x in range(1,11)))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```