Designing a flexible list

Madhavan Mukund

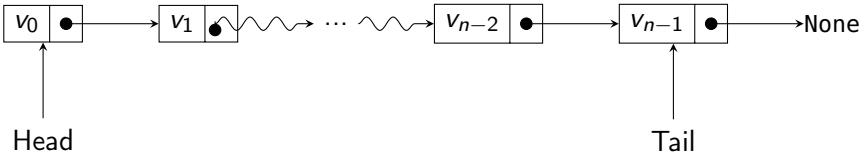
https://www.cmi.ac.in/~madhavan

Programming, Data Structures and Algorithms using Python

Week 3

Lists

- Typically a sequence of nodes
- Each node contains a value and points to the next node in the sequence
  - "Linked" list
- Easy to modify
  - Inserting and deletion is easy via local "plumbing"
  - Flexible size
- Need to follow links to access `A[i]`
  - Takes time $O(i)$



Implementing lists in Python

- Python class `Node`

```
class Node:
    def __init__(self, v = None):
        self.value = v
        self.next = None
        return

    def isempty(self):
        if self.value == None:
            return(True)
        else:
            return(False)
```

Implementing lists in Python

- Python class `Node`
- A list is a sequence of nodes
  - `self.value` is the stored value
  - `self.next` points to next node

```
class Node:
    def __init__(self, v = None):
        self.value = v
        self.next = None
        return

    def isempty(self):
        if self.value == None:
            return(True)
        else:
            return(False)
```

Implementing lists in Python

- Python class `Node`
- A list is a sequence of nodes
  - `self.value` is the stored value
  - `self.next` points to next node
- Empty list?
  - `self.value` is `None`

```
class Node:
    def __init__(self, v = None):
        self.value = v
        self.next = None
        return

    def isempty(self):
        if self.value == None:
            return(True)
        else:
            return(False)
```

Implementing lists in Python

- Python class `Node`
- A list is a sequence of nodes
  - `self.value` is the stored value
  - `self.next` points to next node
- Empty list?
  - `self.value` is `None`
- Creating lists
  - `l1 = Node()` — empty list
  - `l2 = Node(5)` — singleton list

```
class Node:
    def __init__(self, v = None):
        self.value = v
        self.next = None
        return

    def isempty(self):
        if self.value == None:
            return(True)
        else:
            return(False)
```

Implementing lists in Python

- Python class `Node`
- A list is a sequence of nodes
  - `self.value` is the stored value
  - `self.next` points to next node
- Empty list?
  - `self.value` is `None`
- Creating lists
  - `l1 = Node()` — empty list
  - `l2 = Node(5)` — singleton list
  - `l1.isempty() == True`
  - `l2.isempty() == False`

```
class Node:
    def __init__(self, v = None):
        self.value = v
        self.next = None
        return

    def isempty(self):
        if self.value == None:
            return(True)
        else:
            return(False)
```

Appending to a list

- Add `v` to the end of list `l`
- If `l` is empty, update `l.value` from `None` to `v`
- If at last value, `l.next` is `None`
  - Point `next` at new node with value `v`
- Otherwise, recursively append to rest of list

```
def append(self,v):
    # append, recursive
    if self.isempty():
        self.value = v
    elif self.next == None:
        self.next = Node(v)
    else:
        self.next.append(v)
    return
```

Appending to a list

- Add `v` to the end of list `l`
- If `l` is empty, update `l.value` from `None` to `v`
- If at last value, `l.next` is `None`
  - Point `next` at new node with value `v`
- Otherwise, recursively append to rest of list
- Iterative implementation
  - If empty, replace `l.value` by `v`
  - Loop through `l.next` to end of list
  - Add `v` at the end of the list

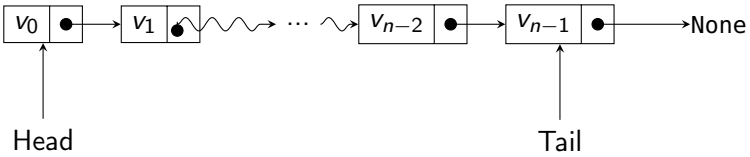
```
def appendi(self,v):
    # append, iterative
    if self.isempty():
        self.value = v
        return

    temp = self
    while temp.next != None:
        temp = temp.next

    temp.next = Node(v)
    return
```

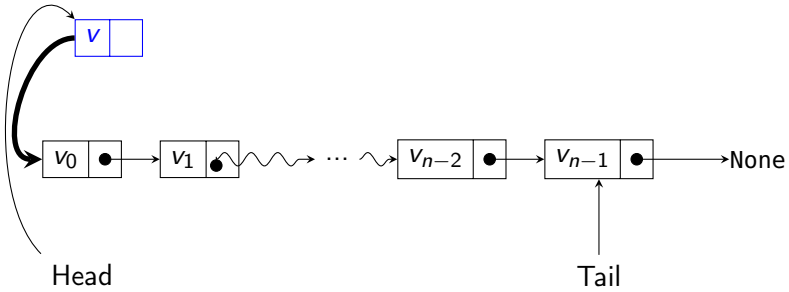
Insert at the start of the list

- Want to insert v at head
- Create a new node with v



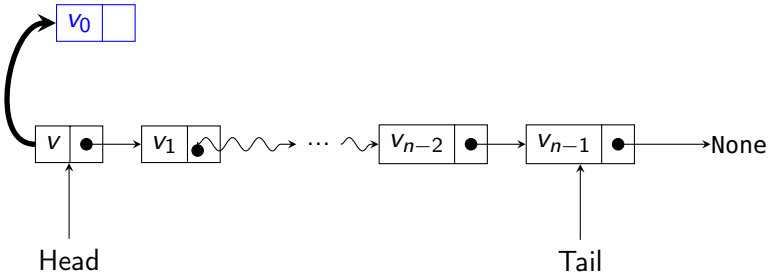
Insert at the start of the list

- Want to insert v at head
- Create a new node with v
- Cannot change where the head points!



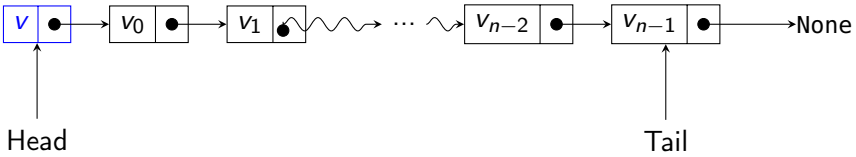
Insert at the start of the list

- Want to insert v at head
- Create a new node with v
- Cannot change where the head points!
- Exchange the values v_0, v



Insert at the start of the list

- Want to insert v at head
- Create a new node with v
- Cannot change where the head points!
- Exchange the values v_0, v
- Make new node point to `head.next`
- Make `head.next` point to new node



Appending to a list

- Create a new node with v
- Exchange the values v_0, v
- Make new node point to `head.next`
- Make `head.next` point to new node

```
def insert(self,v):
    if self.isempty():
        self.value = v
        return

    newnode = Node(v)

    # Exchange values in self and newnode
    (self.value, newnode.value) =
        (newnode.value, self.value)

    # Switch links
    (self.next, newnode.next) =
        (newnode, self.next)

    return
```

Delete a value v

- Remove first occurrence of v
- Scan list for first v — look ahead at next node
- If next node value is v, bypass it

Delete a value v

- Remove first occurrence of v
- Scan list for first v — look ahead at next node
- If next node value is v, bypass it
- Cannot bypass the first node in the list
  - Instead, copy the second node value to head
  - Bypass second node

Delete a value v

- Remove first occurrence of v
- Scan list for first v — look ahead at next node
- If next node value is v, bypass it
- Cannot bypass the first node in the list
  - Instead, copy the second node value to head
  - Bypass second node
- Recursive implementation

```
def delete(self,v):
# delete, recursive

    if self.isempty():
        return

    if self.value == v:
        self.value = None
        if self.next != None:
            self.value = self.next.value
            self.next = self.next.next
        return
    else:
        if self.next != None:
            self.next.delete(v)
            if self.next.value == None:
                self.next = None
    return
```

Delete a value v

- Remove first occurrence of v
- Scan list for first v — look ahead at next node
- If next node value is v, bypass it
- Cannot bypass the first node in the list
  - Instead, copy the second node value to head
  - Bypass second node
- Recursive implementation
- Exercise: write an iterative version

```
def delete(self,v):
# delete, recursive

    if self.isempty():
        return

    if self.value == v:
        self.value = None
        if self.next != None:
            self.value = self.next.value
            self.next = self.next.next
        return
    else:
        if self.next != None:
            self.next.delete(v)
            if self.next.value == None:
                self.next = None
    return
```

Summary

- Use a linked list of nodes to implement a flexible list
- Append is easy
- Insert requires some care, cannot change where the head points to
- When deleting, look one step ahead to bypass the node to be deleted