Advanced Topics

Lists and Dynamic Programming

Brad Miller David Ranum

1/25/06

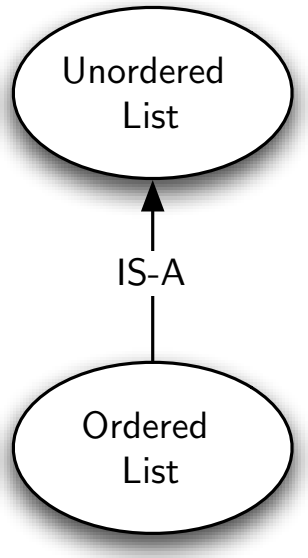
Outline

Outline

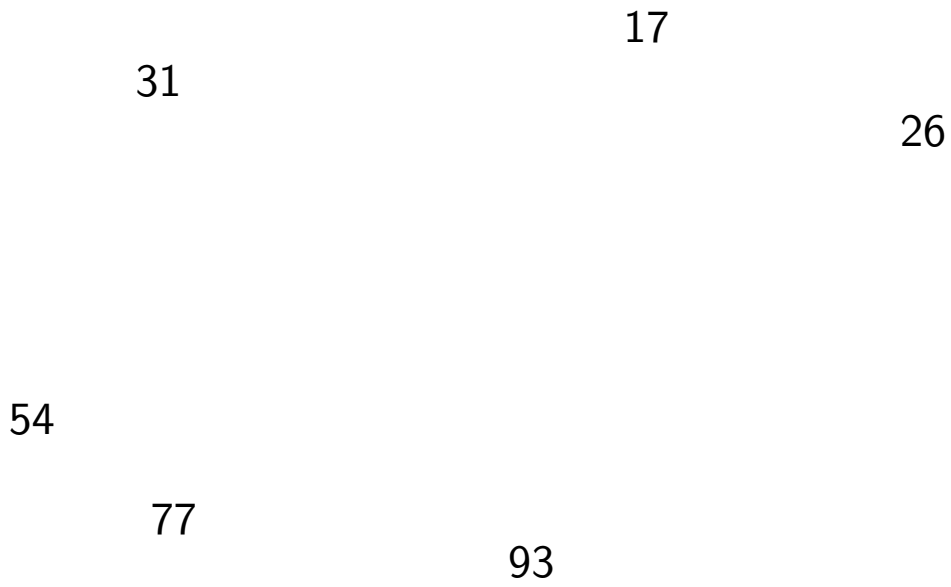
- `List()` creates a new list that is empty. It needs no parameters and returns an empty list.
- `add(item)` adds a new item to the list. It needs the item and returns nothing. Assume the item is not already in the list.
- `remove(item)` removes the item from the list. It needs the item and modifies the list. Assume the item is present in the list.
- `search(item)` searches for the item in the list. It needs the item and returns a boolean value.
- `isEmpty()` tests to see whether the list is empty. It needs no parameters and returns a boolean value.
- `length()` returns the number of items in the list. It needs no parameters and returns an integer.

Outline

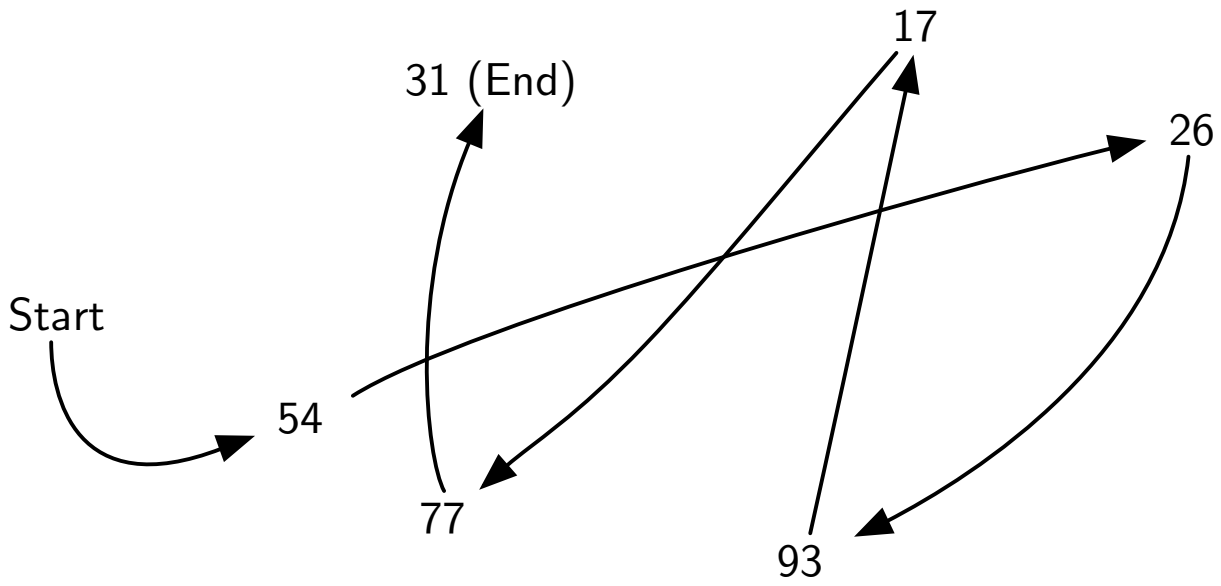
Hierarchical Relationship Between Unordered and Ordered Lists



Items Not Constrained in Their Physical Placement



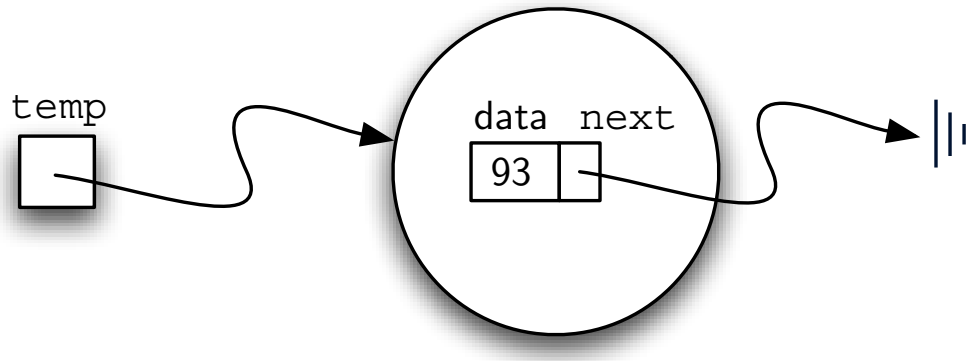
Relative Positions Maintained by Explicit Links.



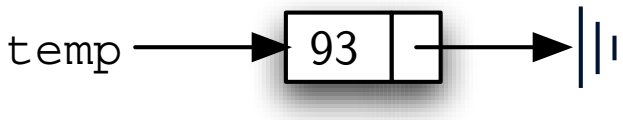
A Node Class

```
1  class Node:
2      def __init__(self,initdata):
3          self.data = initdata
4          self.next = None
5
6      def getData(self):
7          return self.data
8
9      def getNext(self):
10         return self.next
11
12     def setData(self,newdata):
13         self.data = newdata
14
15     def setNext(self,newnext):
16         self.next = newnext
```

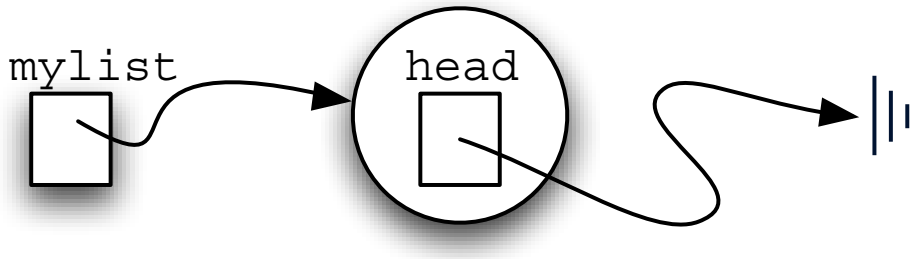
A Node Object Contains the Item and a Reference to the Next Node



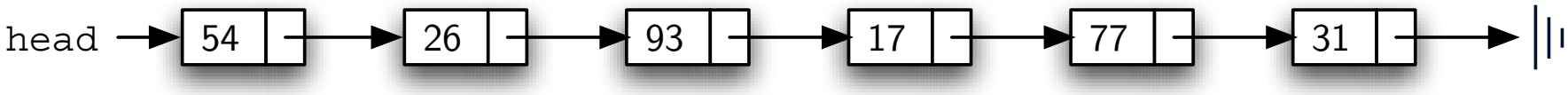
A Typical Representation for a Node



An Empty List



A Linked List of Integers



The `UnorderedList` Class Constructor

```
1  class UnorderedList:
2      def __init__(self):
3          self.head = None
```

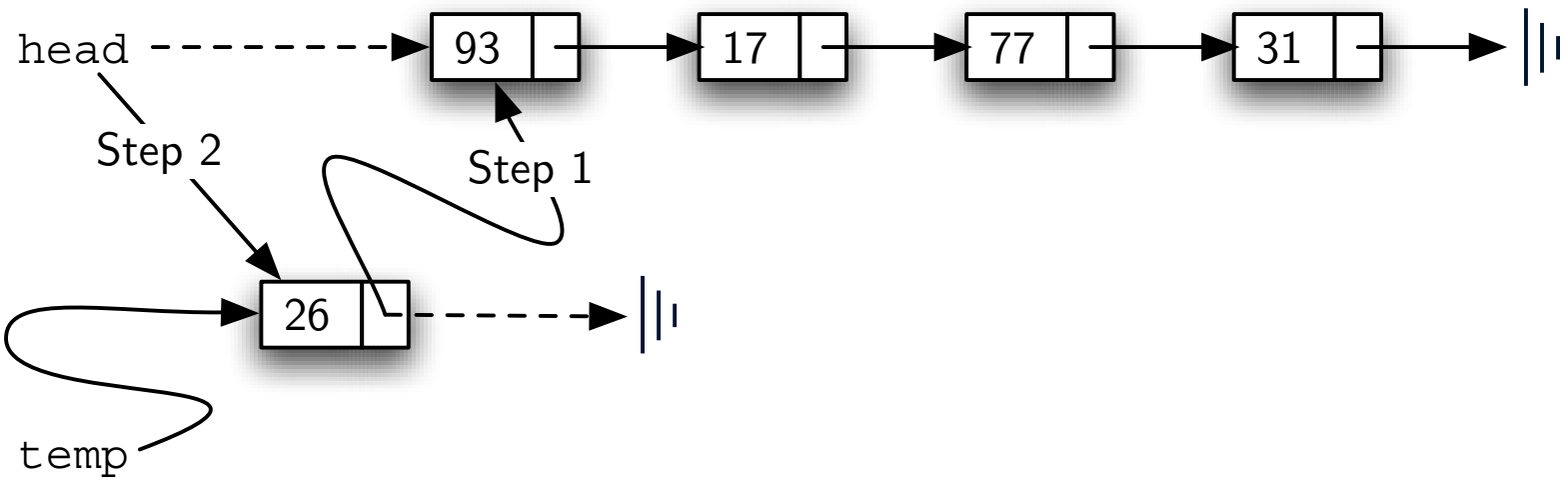
The `isEmpty` Method

```
1  def isEmpty(self):
2      return self.head == None
```

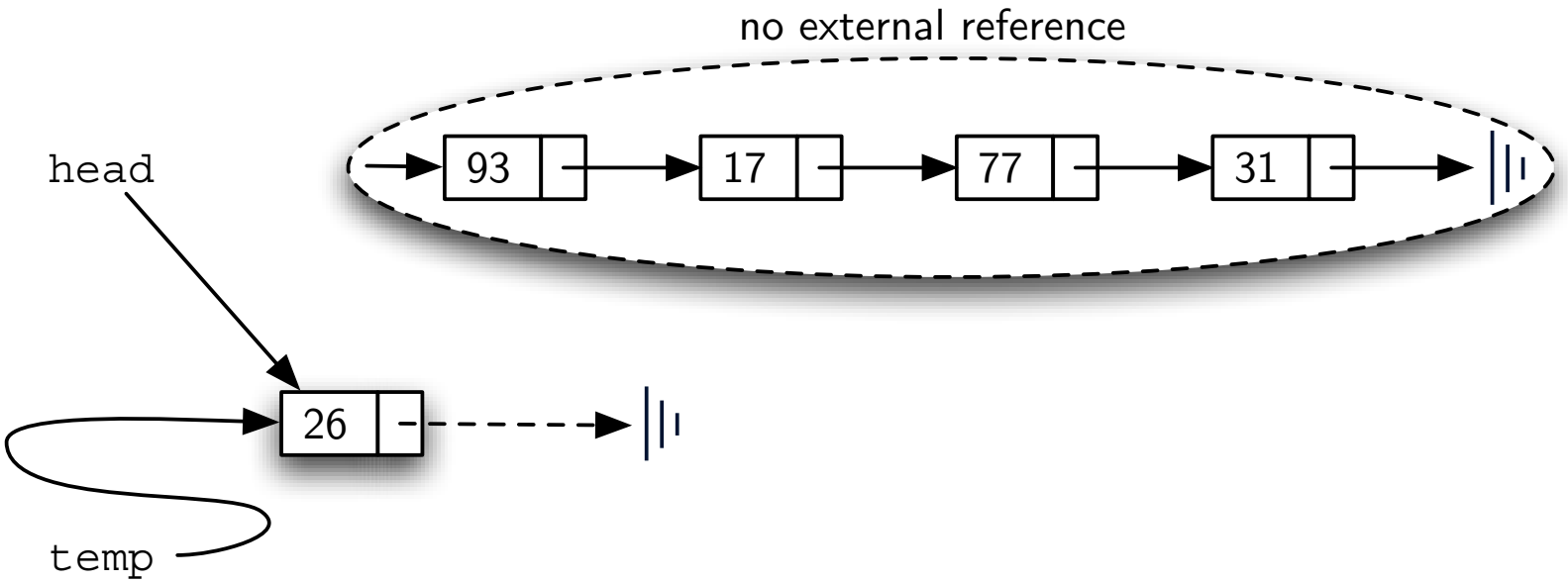
The `add` Method

```
1  def add(self,item):
2      temp = Node(item)
3      temp.setNext(self.head)
4      self.head = temp
```

Adding a New Node is a Two-Step Process



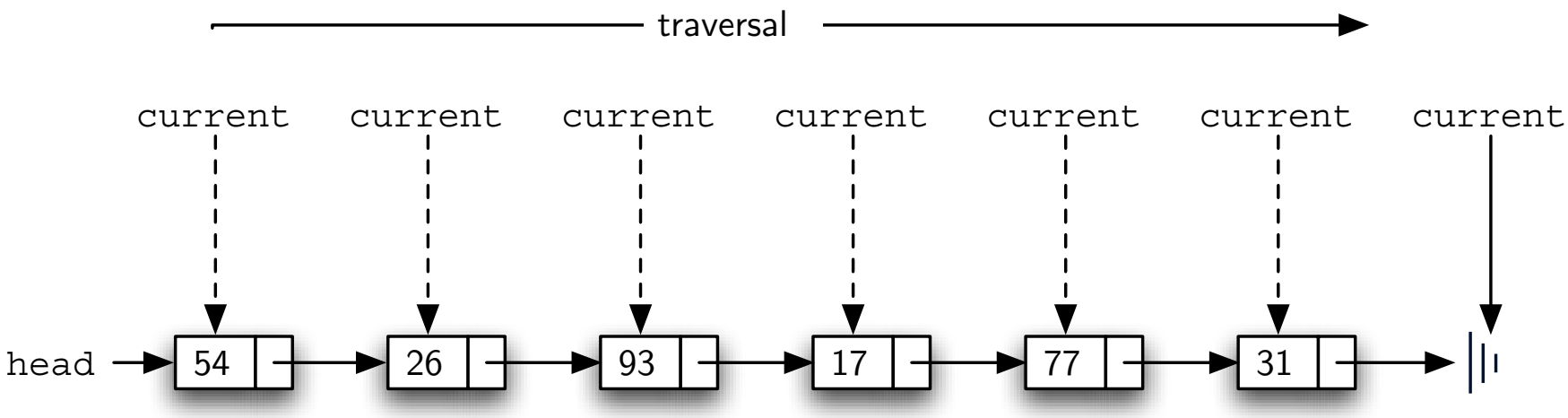
Result of Reversing the Order of the Two Steps



The `length` Method

```
1  def length(self):
2      current = self.head
3      count = 0
4      while current != None:
5          count = count + 1
6          current = current.getNext()
7
8      return count
```

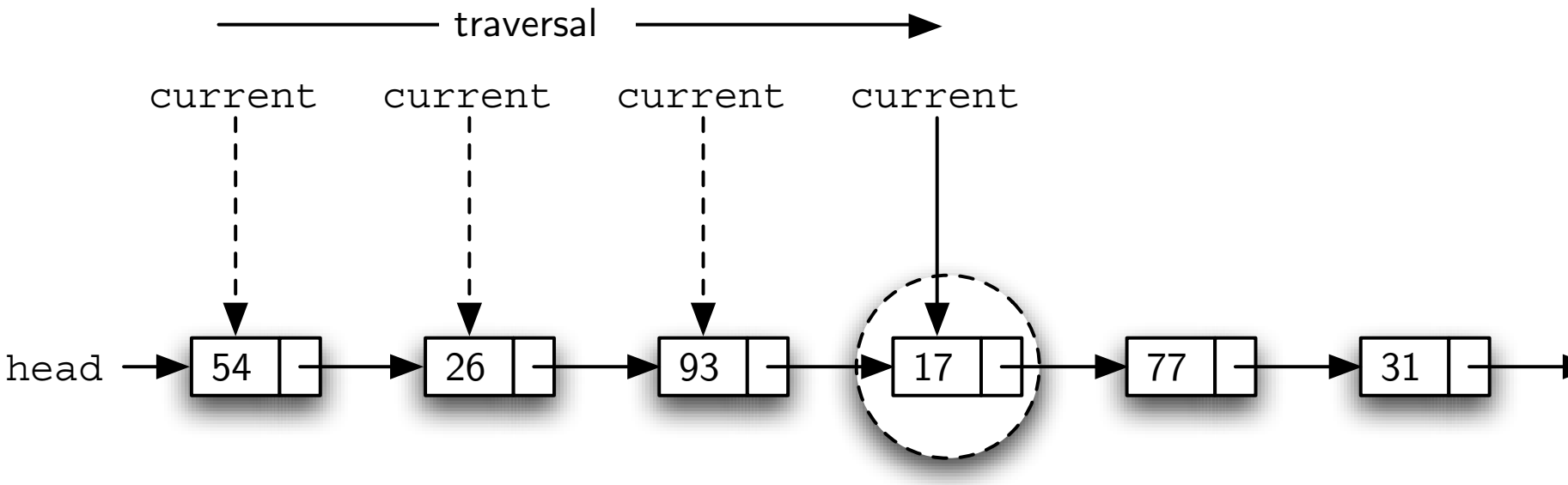
Traversing the Linked List from the Head to the End



The `search` Method

```
1  def search(self,item):
2      current = self.head
3      found = False
4      while current != None and not found:
5          if current.getData() == item:
6              found = True
7          else:
8              current = current.getNext()
9
10     return found
```

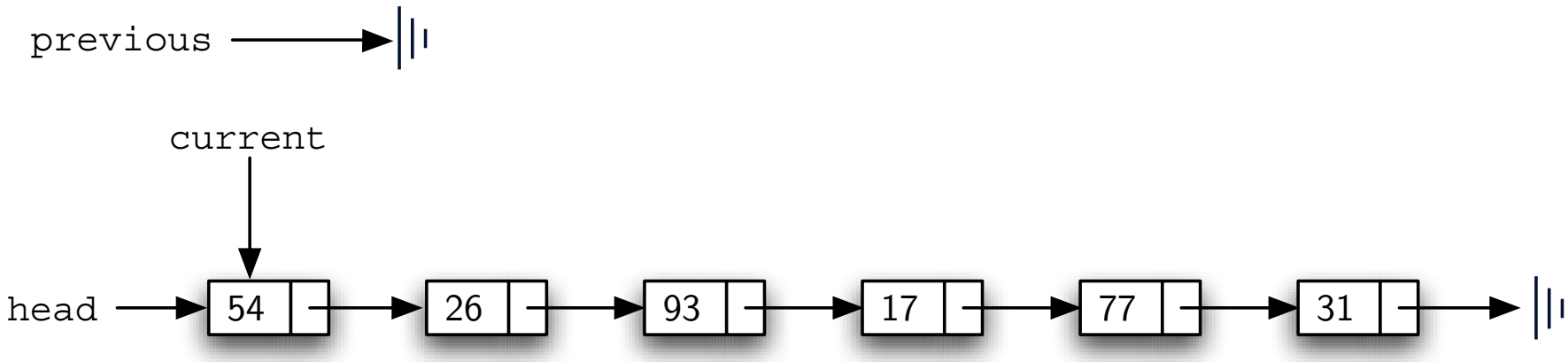
Successful Search for the Value 17



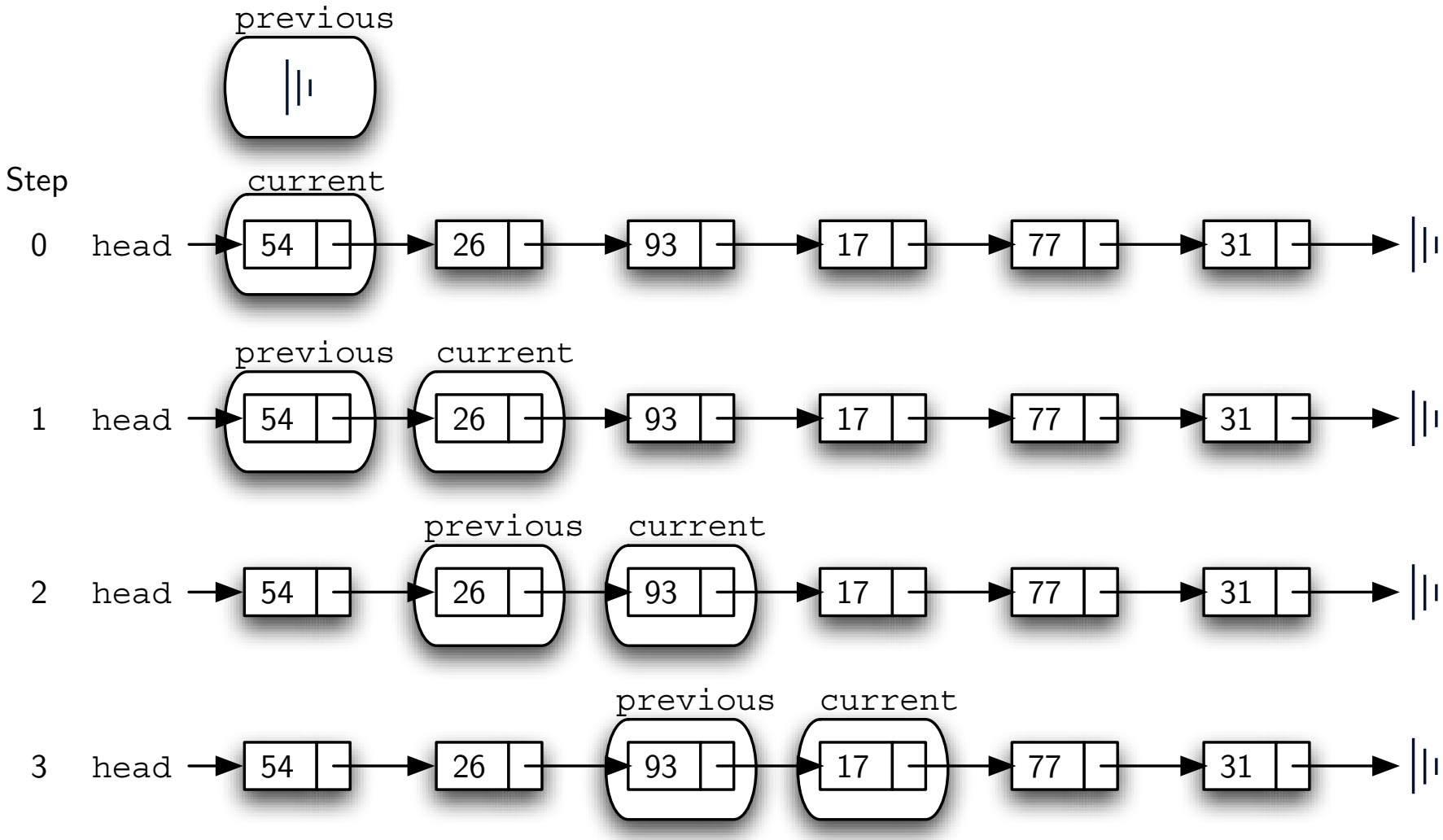
The `remove` Method

```
1  def remove(self,item):
2      current = self.head
3      previous = None
4      found = False
5      while not found:
6          if current.getData() == item:
7              found = True
8          else:
9              previous = current
10             current = current.getNext()
11
12     if previous == None:
13         self.head = current.getNext()
14     else:
15         previous.setNext(current.getNext())
```

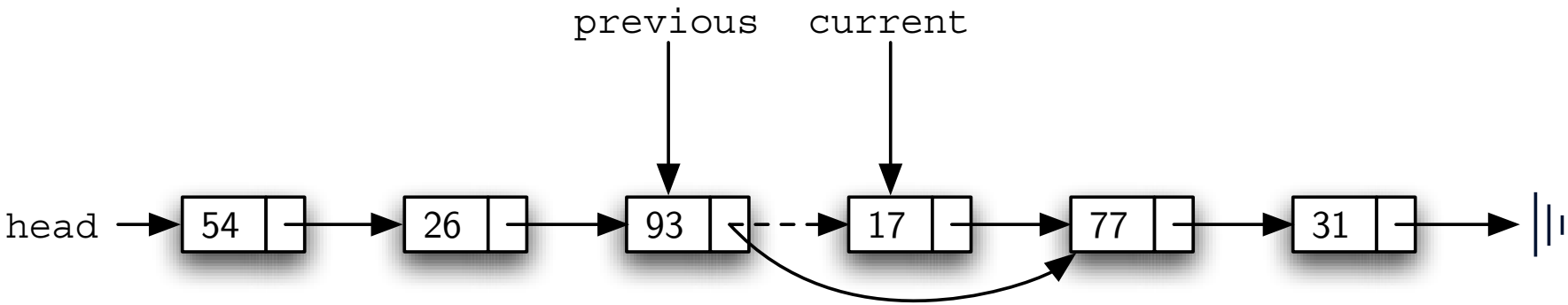
Initial Values for the `previous` and `current` References



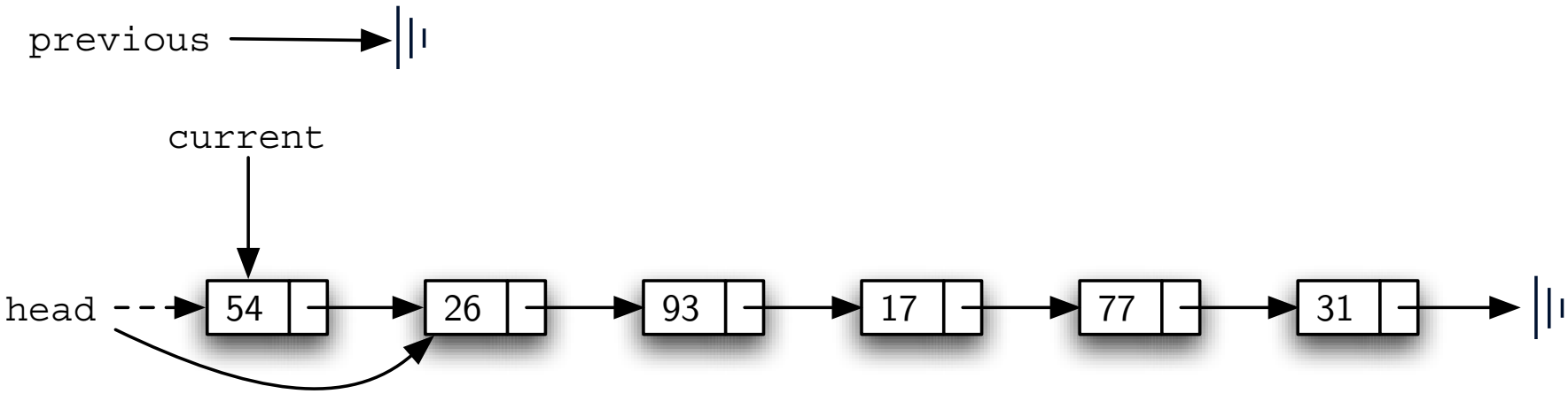
previous and current Move Down the List



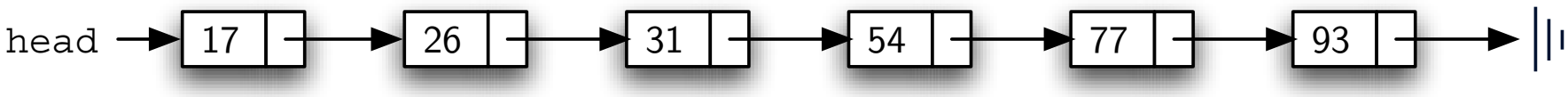
Removing an Item from the Middle of the List



Removing the First Node from the List



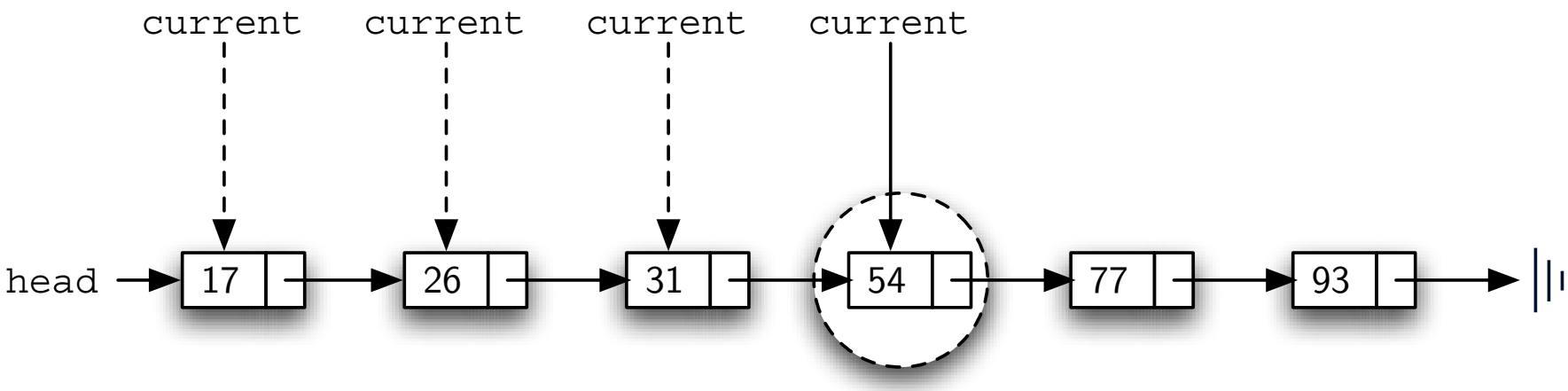
An Ordered Linked List



`OrderedList` Inherits from `UnorderedList`

```
1  class OrderedList(UnorderedList):
2      def __init__(self):
3          UnorderedList.__init__(self)
```

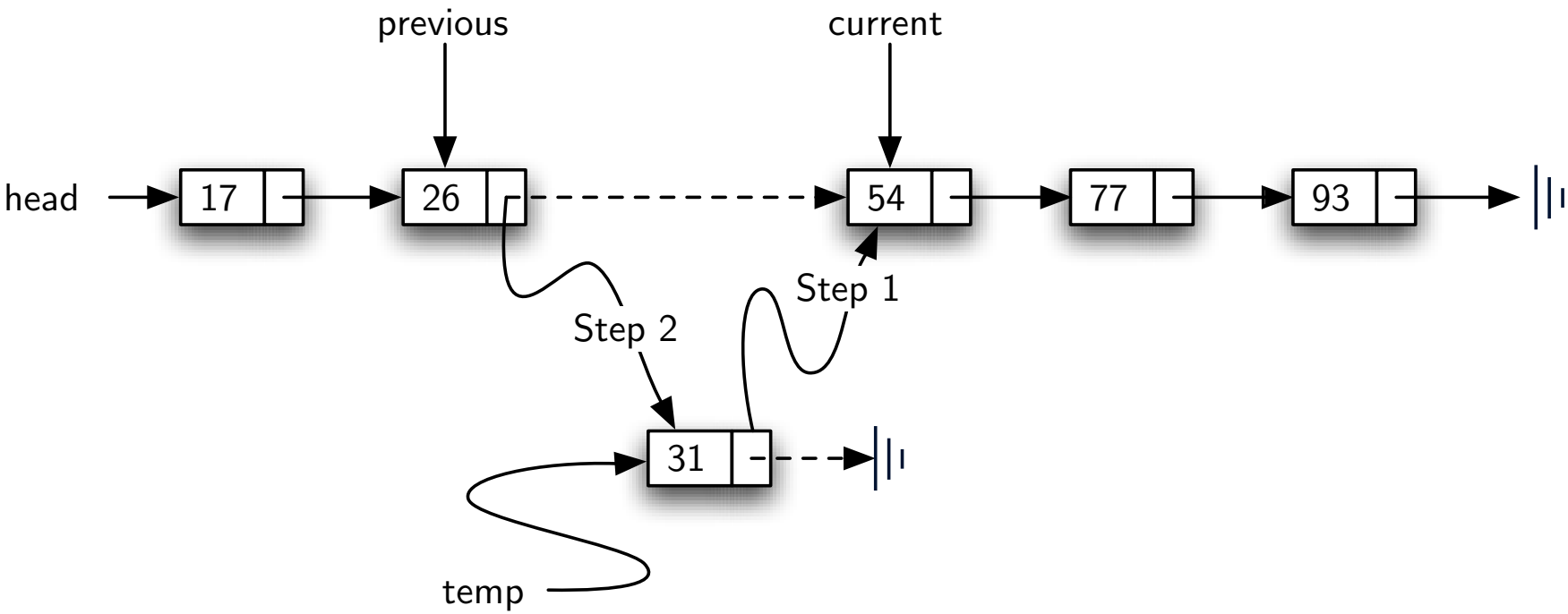
Searching an Ordered Linked List



The Modified `search` Method for the Ordered List

```
1   def search(self,item):
2       current = self.head
3       found = False
4       stop = False
5       while current != None and not found and not stop:
6           if current.getData() == item:
7               found = True
8           else:
9               if current.getData() > item:
10                  stop = True
11              else:
12                  current = current.getNext()
13
14      return found
```

Adding an Item to an Ordered Linked List



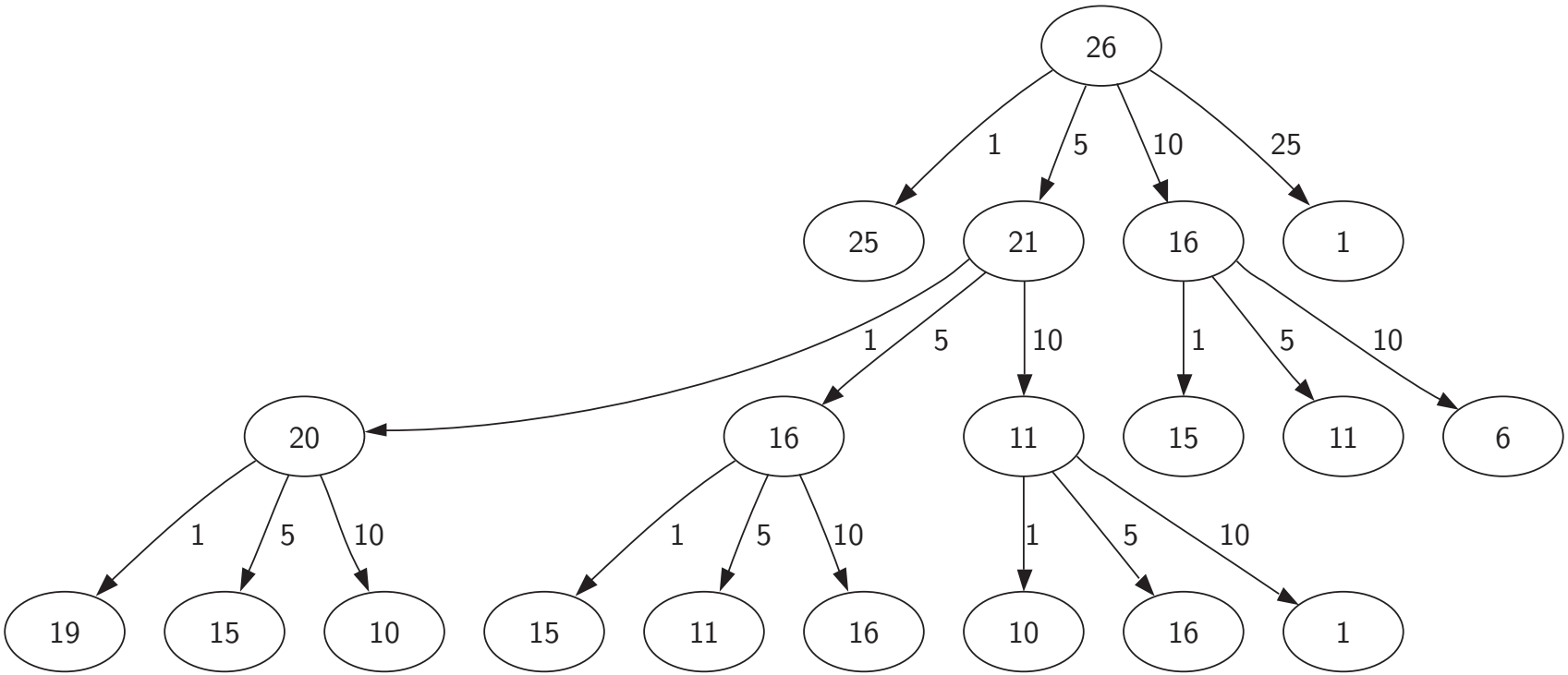
The Modified `add` Method for the Ordered List

```
1  def add(self,item):
2      current = self.head
3      previous = None
4      stop = False
5      while current != None and not stop:
6          if current.getData() > item:
7              stop = True
8          else:
9              previous = current
10             current = current.getNext()
11     temp = Node(item)
12     if previous == None:
13         temp.setNext(self.head)
14         self.head = temp
15     else:
16         temp.setNext(current)
17         previous.setNext(temp)
```

Recursive Version of Coin Optimization Problem

```
1  def recMC(coins,change):
2      minCoins = change
3      if change in coins:
4          return 1
5      else:
6          for i in [c for c in coins if c <= change]:
7              numCoins = 1 + recMC(coins,change-i)
8              if numCoins < minCoins:
9                  minCoins = numCoins
10     return minCoins
```

Call Tree.

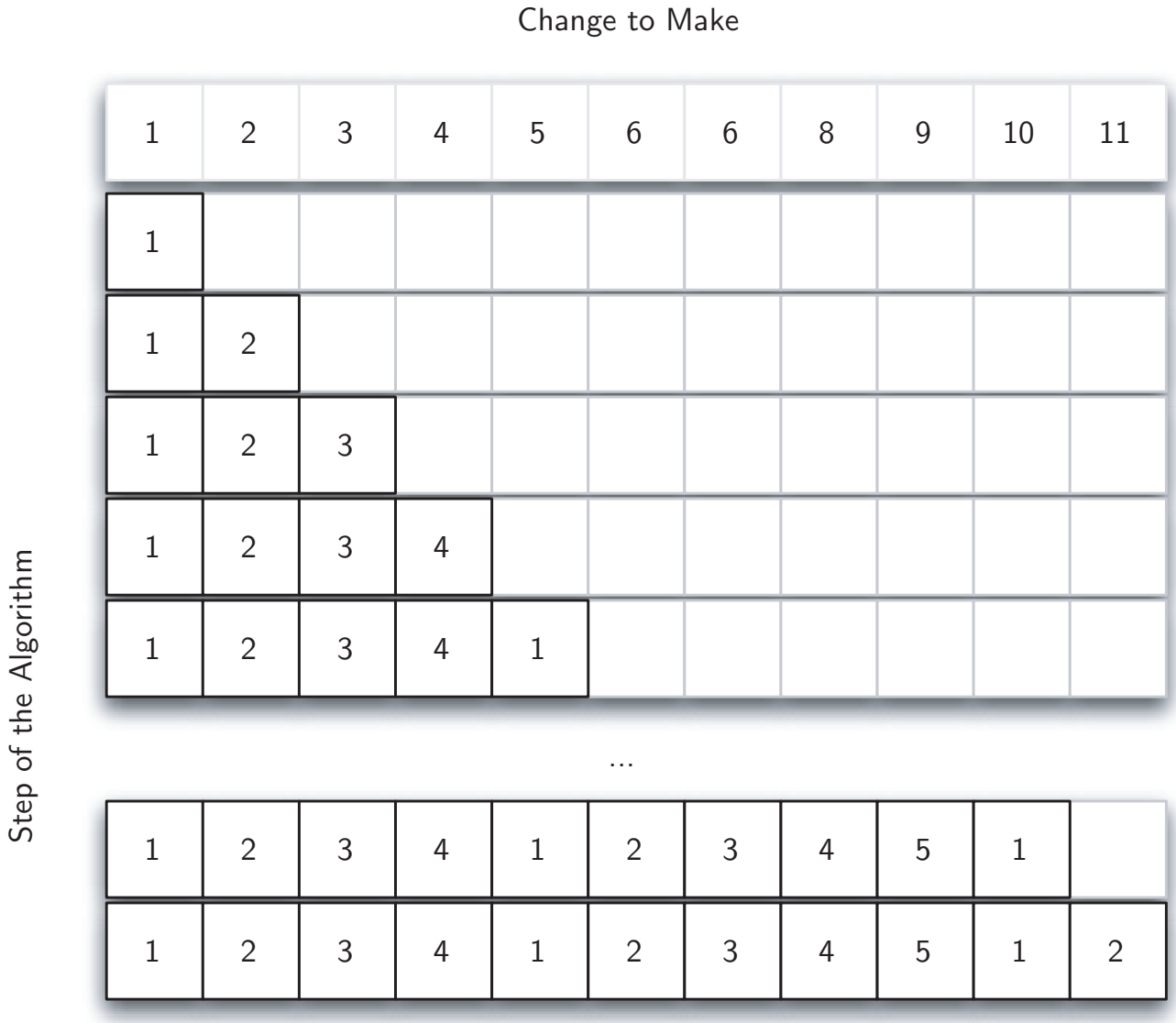

26
1
5
10
25
25
21
16
1
1
5
10
1
5
10
20
16
11
15
11
6
1
5
10
1
5
10
1
5
10
19
15
10
15
11
16
10
16
1

Recursive Coin Optimization Using Table Lookup

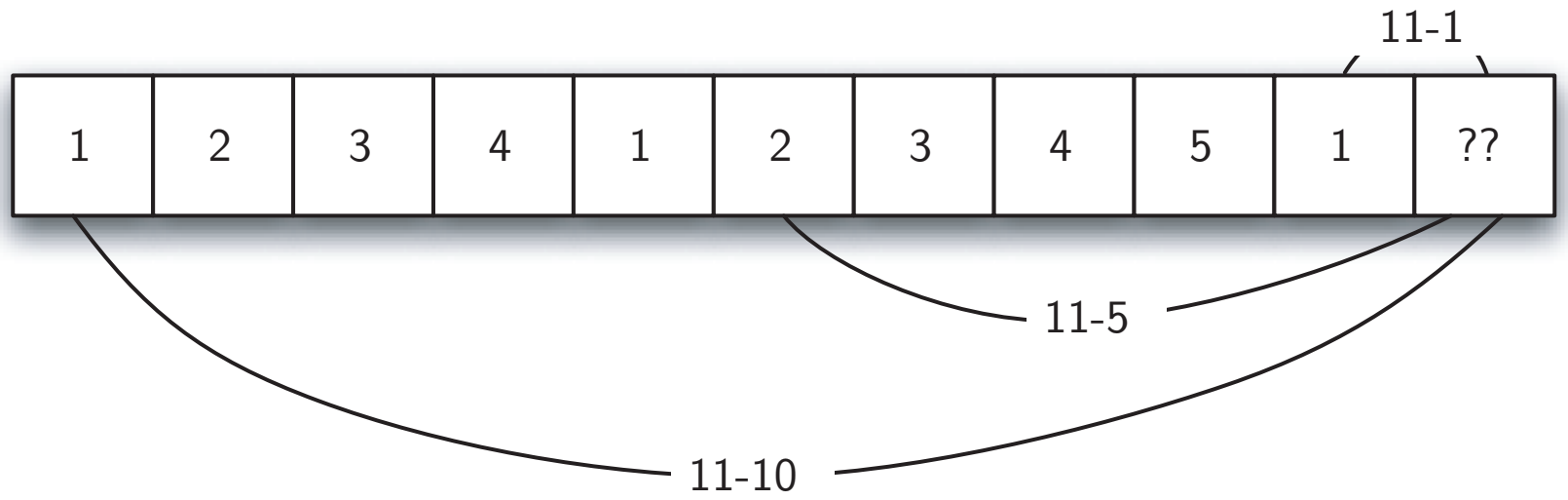
```
1  def recDC(coins,change,res):
2      minCoins = change
3      if change in coins:
4          res[change] = 1
5          return 1
6      elif res[change] > 0:
7          return res[change]
8      else:
9          for i in [c for c in coins if c <= change]:
10             numCoins = 1 + recDC(coins,change-i,res)
11             if numCoins < minCoins:
12                 minCoins = numCoins
13                 res[change] = minCoins
14     return minCoins
```

1. A penny plus the minimum number of coins to make change for $11 - 1 = 10$ cents (1)
2. A nickel plus the minimum number of coins to make change for $11 - 5 = 6$ cents (2)
3. A dime plus the minimum number of coins to make change for $11 - 10 = 1$ cent (1)

Minimum Number of Coins Needed to Make Change



Three Options to Consider for the Minimum Number of Coins for Eleven Cents



Dynamic Programming Solution

```
1  def dpMakeChange(coinList,change,minCoins):
2      for cents in range(change+1):
3          coinCount = cents
4          for j in [c for c in coinList if c <= cents]:
5                  if minCoins[cents-j] + 1 < coinCount:
6                      coinCount = minCoins[cents-j]+1
7          minCoins[cents] = coinCount
8      return minCoins[change]
```

Modified Dynamic Programming Solution I

```
1  def dpMakeChange(coinList,change,minCoins,coinsUsed):
2      for cents in range(change+1):
3          coinCount = cents
4          newCoin = 1
5          for j in [c for c in coinList if c <= cents]:
6              if minCoins[cents-j] + 1 < coinCount:
7                  coinCount = minCoins[cents-j]+1
8                  newCoin = j
9          minCoins[cents] = coinCount
10         coinsUsed[cents] = newCoin
11     return minCoins[change]
12
13
14
15
16
17
```

Modified Dynamic Programming Solution II

```
18  def printCoins(coinsUsed,change):
19      coin = change
20      coinDict = {}
21      while coin > 0:
22          thisCoin = coinsUsed[coin]
23          print thisCoin
24          coin = coin - thisCoin
```