ECS 32B - Linear Data Structures

Aaron Kaloti

UC Davis - Summer Session #2 2020

Overview

- What are is a linear data structure?
- Stacks.
- Queues.
- Deques.
- Lists (unordered lists and ordered lists).
    - Linked lists.
    - Unsorted lists vs. sorted lists.
- ADTs (abstract data types) vs. how they're implemented.

Linear Data Structure

Definitions

- **data structure**[1]: way to store/organize data to facilitate access and modifications.
  - Some data structures are better in different scenarios; important to know their strengths.
- **linear data structure**[1]: elements are ordered based on how added/removed.
  - In this course: stacks, queues, deques, lists.
  - Can view as having two ends (with arbitrary names such as left/right, front/rear, top/bottom, top/base).
  - Where elements are added/removed:

Structure	Add Element	Remove Element
Stack	Left (usu. top)	Left (usu. top)
Queue	Left (usu. front)	Right (usu. rear)
Deque	Either end	Either end
List	Anywhere	Anywhere

1. Source: p.9 of *Introduction to Algorithms* by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein (Third Edition).

2. Source: Section 4.2 of *Problem Solving with Algorithms and Data Structures using Python* by Brad Miller and David Ranum.

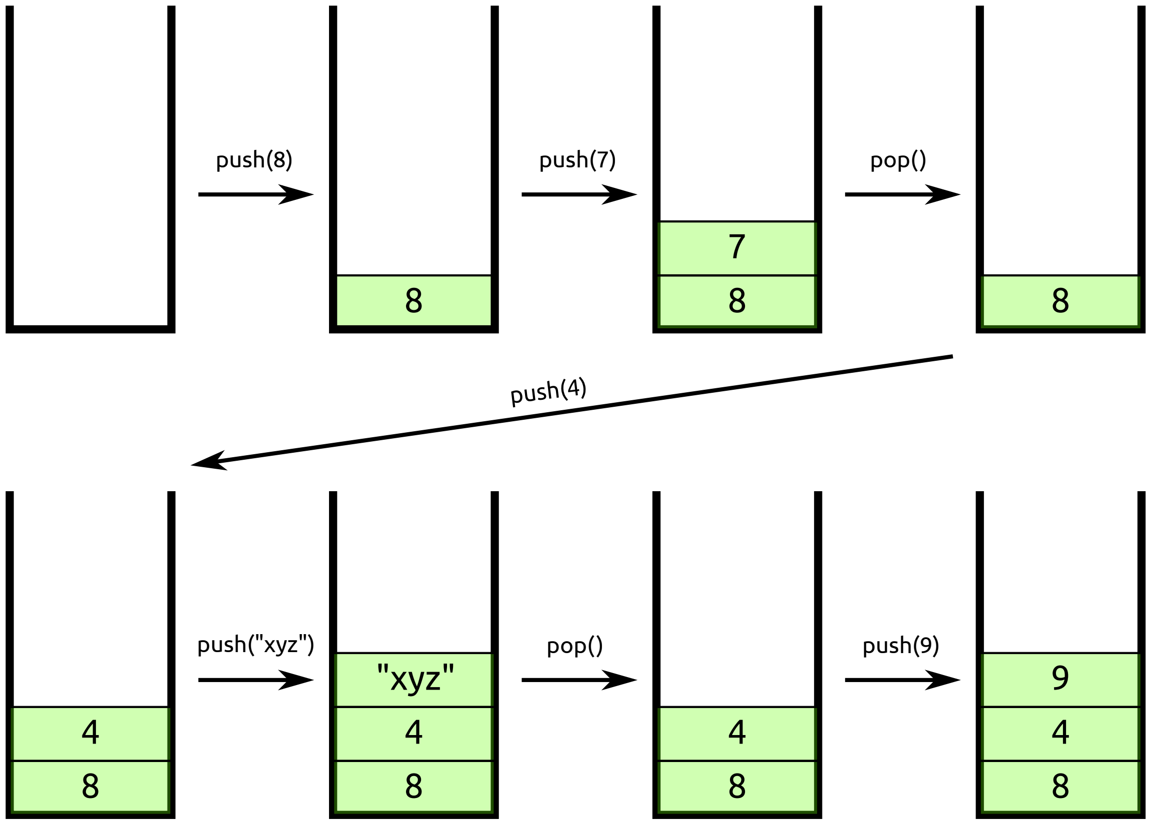
Stack

Overview

- "[A]ddition of new items and the removal of existing items always takes place at the same end"[1].
- LIFO, last-in first-out.

- Two *core* operations:
  1. `push(elem)`: Add `elem` to top.
  2. `pop()`: Remove/retrieve the element at the top.

Example

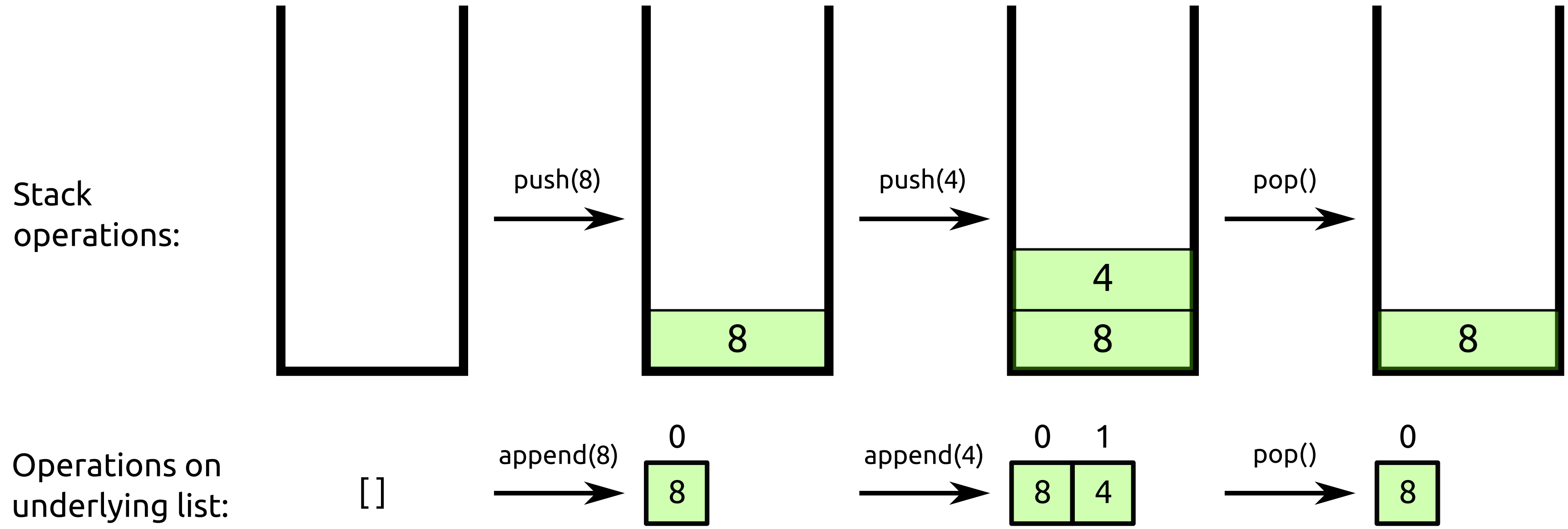


1. Soruce: Section 4.3 of *Problem Solving with Algorithms and Data Structures using Python* by Brad Miller and David Ranum.

Stack

Implementation: Prerequisite

- Underlying implementation will be a Python list.
- Important list methods (for both stacks *and* queues)[1]:
    - `append(elem)`: adds `elem` to end of list.
    - `pop(index=-1)`: removes element at `index`.
        - `index` is optional; `pop()` removes last element.
    - `index(elem)`: returns index of first occurrence of `elem`.



1. Further reading: https://www.w3schools.com/python/python_ref_list.asp

Stack

Textbook's Implementation[1]

```python
class Stack:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def peek(self):
        return self.items[len(self.items)-1]

    def size(self):
        return len(self.items)
```

stack.py

- Each of these methods takes constant time.

1. From section 4.5 of *Problem Solving with Algorithms and Data Structures using Python* by Brad Miller and David Ranum.

Stack

Example: Balanced Parentheses

Prompt

- Write a function called `parChecker()`[1] that takes as argument a string of opening and closing parentheses and returns true if the parentheses are balanced, i.e. "each opening symbol has a corresponding closing symbol and the pairs of parentheses are properly nested."
- Write unit tests for this function.

Examples

- Balanced:
  - `(()()()())`
  - `(((())))`
  - `(()((())()))`
- Unbalanced:
  - `((((((())`
  - `()))`
  - `(()()(()`

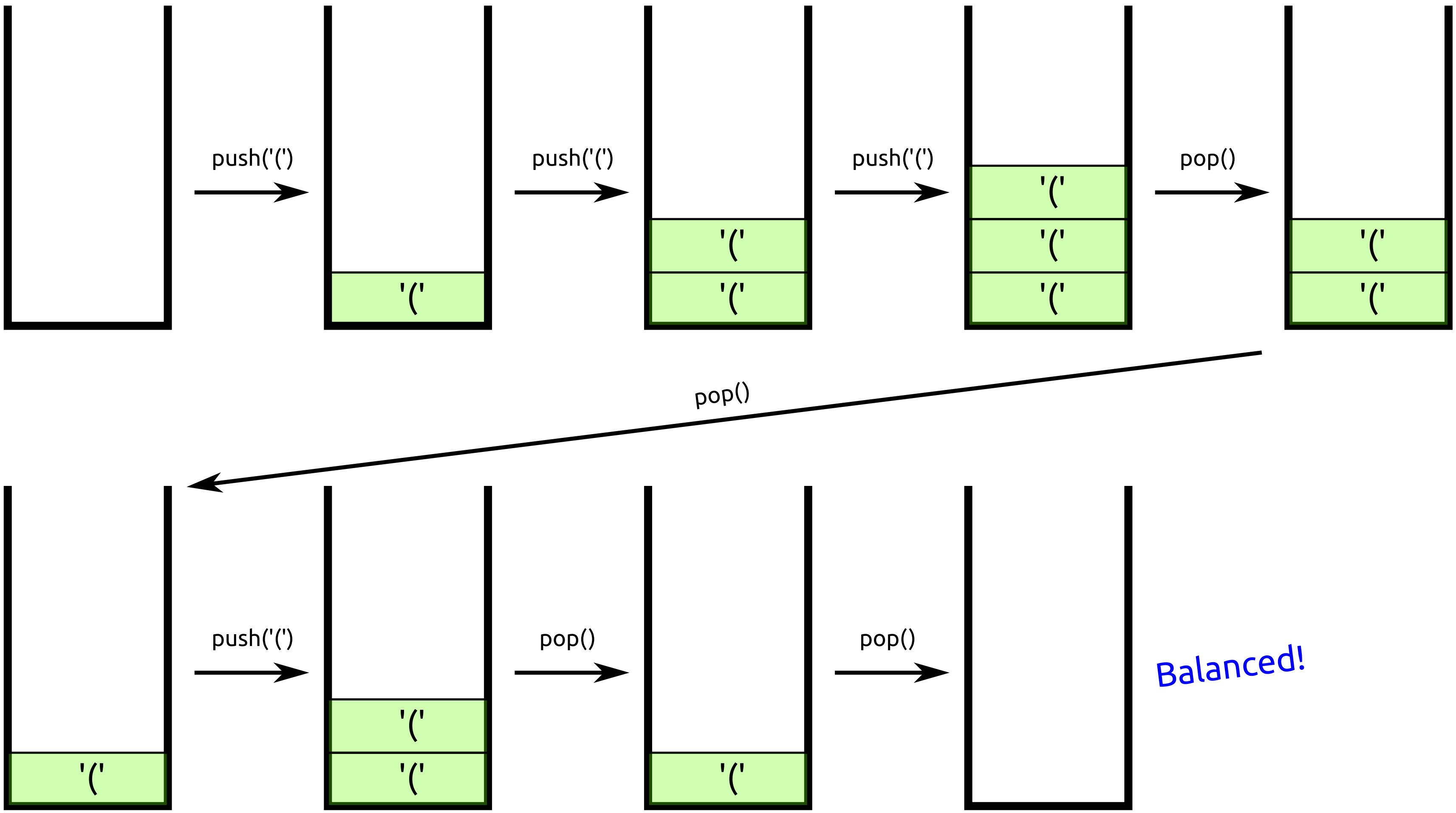
1. I dislike camel case (and prefer snake case) when naming Python functions, but this example is from section 4.6 of *Problem Solving with Algorithms and Data Structures using Python* by Brad Miller and David Ranum, so we're keeping their naming convention.

Stack

Example: Balanced Parentheses

Scenario #1: Balanced

String: "((())(()))"

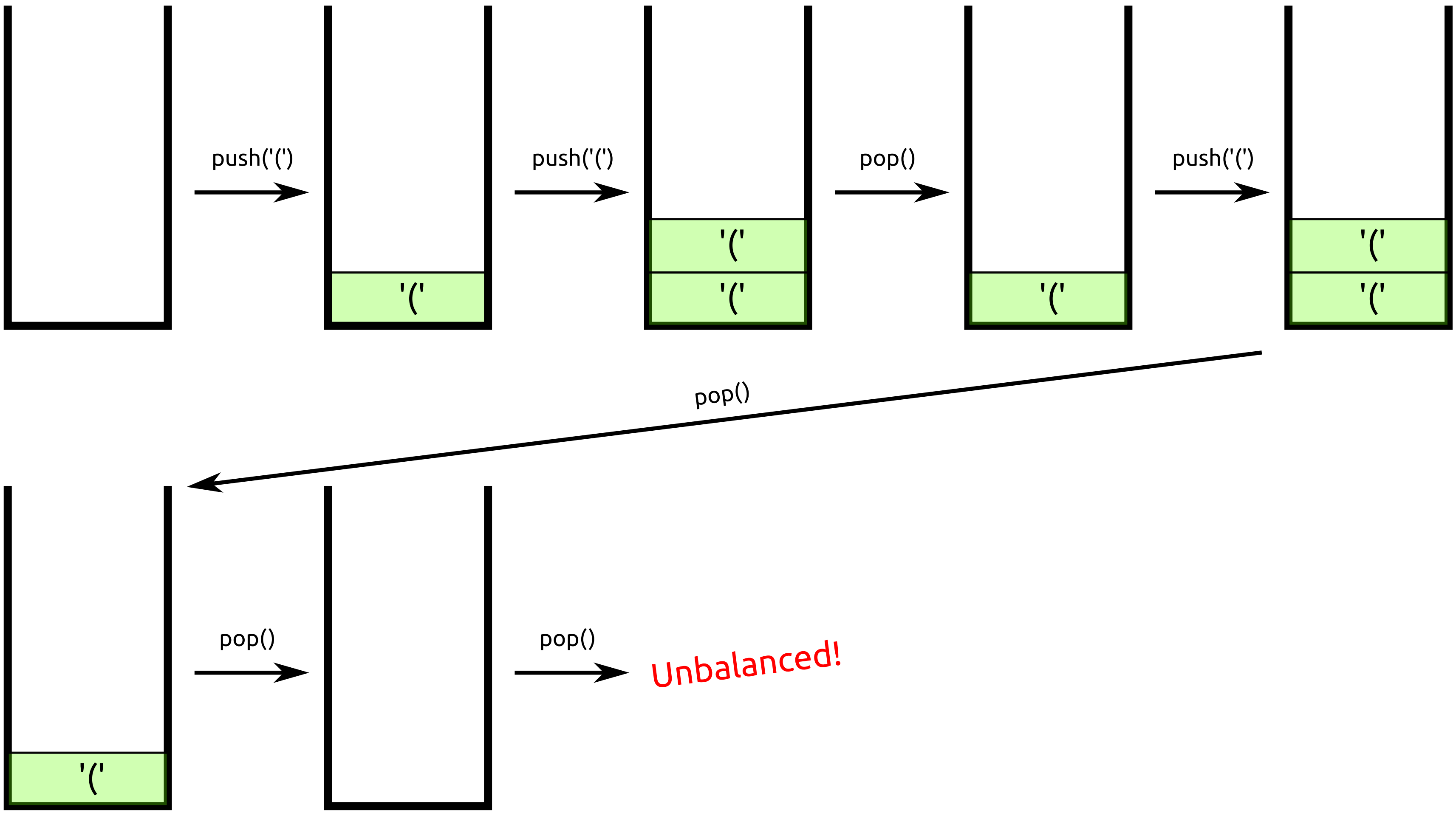


Stack

Example: Balanced Parentheses

Scenario #2: Unbalanced

String: "(()()))(())"

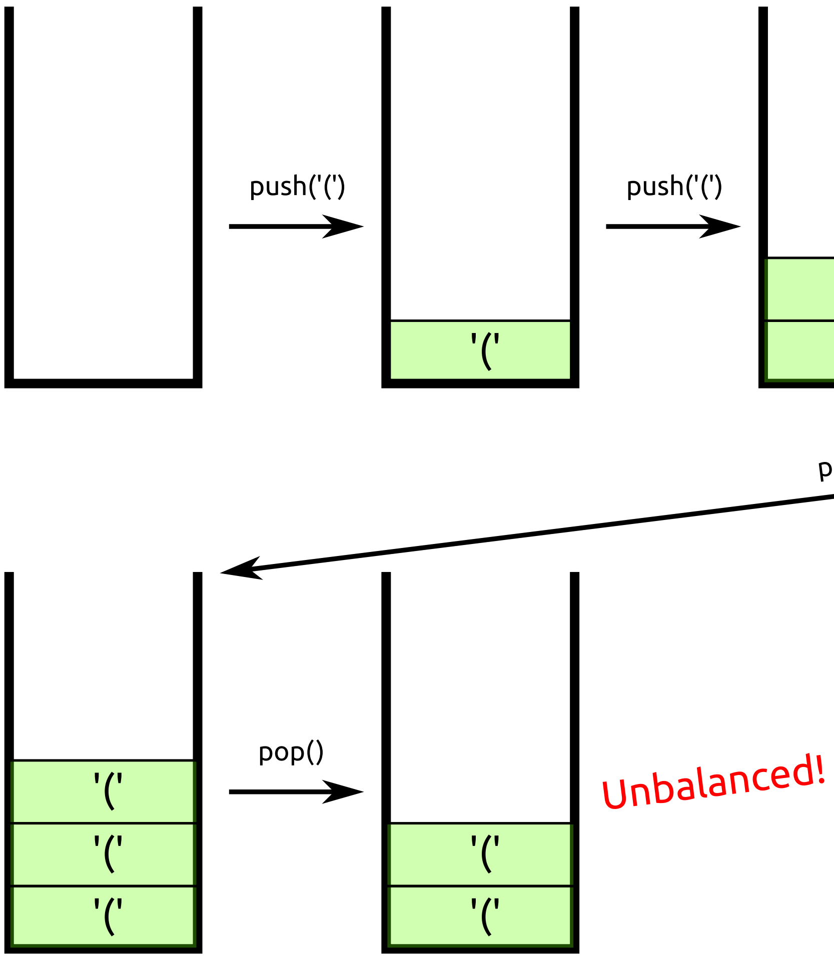


Stack

Example: Balanced Parentheses

Scenario #3: Unbalanced (in a Different Way)

String: "(((())"



Stack

Example: Balanced Parentheses

Unit Tests

```python
import unittest
from par_checker import parChecker

class TestParChecker(unittest.TestCase):
    def test_case1(self):
        inp = "(()()()())"
        self.assertTrue(parChecker(inp))

    def test_case2(self):
        inp = "(((())))"
        self.assertTrue(parChecker(inp))

    def test_case3(self):
        inp = "((((((())"
        self.assertFalse(parChecker(inp))

    def test_case4(self):
        inp = "()))"
        self.assertFalse(parChecker(inp))

    def test_case5(self):
        inp = "(()()(()"
        self.assertFalse(parChecker(inp))

if __name__ == "__main__":
    unittest.main()
```

test_par_checker.py

Stack

Example: Balanced Parentheses

My Solution: `parChecker()`

```
from stack import Stack

def parChecker(s):
    unmatched_parentheses = Stack()
    for c in s:
        if c == '(':
            unmatched_parentheses.push(c)
        else:  # c == ')'
            try:
                unmatched_parentheses.pop()
            except IndexError:
                return False
    return unmatched_parentheses.isEmpty()
##    if stack.isEmpty():
##        return True
##    else:
##        return False
```

Stack

Example: Balanced Parentheses

Textbook's Solution

- Assumes `Stack` definition is in `stack.py`.

```
# from pythonds.basic import Stack
from stack import Stack

def parChecker(symbolString):
    s = Stack()
    balanced = True
    index = 0
    while index < len(symbolString) and balanced:
        symbol = symbolString[index]
        if symbol == "(":
            s.push(symbol)
        else:
            if s.isEmpty():
                balanced = False
            else:
                s.pop()

        index = index + 1

    if balanced and s.isEmpty():
        return True
    else:
        return False

# print(parChecker('((()))'))
# print(parChecker('(()'))
```

Stack

Example: Balanced Symbols

- You should read section 4.7 of *Problem Solving with Algorithms and Data Structures using Python* by Brad Miller and David Ranum on your own time.
  - As stated in the syllabus, the book is freely available.

Stack

Example: Postfix Evaluation

Prompt

- Write a function that takes a postfix expression (string) and returns the result. Write unit tests.

Definitions

- **infix notation**: operator is *between* its two operands. Normal way.
- **prefix notation**: operator is *before* its two operands.
- **postfix notation**: operator is *after* its two operands.

Examples[1]

Infix Notation	Prefix Notation	Postfix Notation
$A + B$	$+ A B$	$A B +$
$A + B \cdot C$	$+ A \cdot B C$	$A B C \cdot +$
$(A + B) \cdot C$	$\cdot + A B C$	$A B + C \cdot$
$A + B \cdot C + D$	$+ + A \cdot B C D$	$A B C \cdot + D +$
$(A + B) \cdot (C + D)$	$\cdot + A B + C D$	$A B + C D + \cdot$
$A \cdot B + C \cdot D$	$+ \cdot A B \cdot C D$	$A B \cdot C D \cdot +$
$A + B + C + D$	$+ + + A B C D$	$A B + C + D +$

1. From section 4.9 of *Problem Solving with Algorithms and Data Structures using Python* by Brad Miller and David Ranum.

Stack

Example: Postfix Evaluation

Textbook's Implementation[1]

```
# from pythonds.basic import Stack
from stack import Stack

def postfixEval(postfixExpr):
    operandStack = Stack()
    tokenList = postfixExpr.split()

    for token in tokenList:
        if token in "0123456789":
            operandStack.push(int(token))
        else:
            operand2 = operandStack.pop()
            operand1 = operandStack.pop()
            result = doMath(token,operand1,operand2)
            operandStack.push(result)
    return operandStack.pop()

def doMath(op, op1, op2):
    if op == "*":
        return op1 * op2
    elif op == "/":
        return op1 / op2
    elif op == "+":
        return op1 + op2
    else:
        return op1 - op2

print(postfixEval('7 8 + 3 2 + /'))
```

1. From section 4.9.3 of *Problem Solving with Algorithms and Data Structures using Python* by Brad Miller and David Ranum.

Stack

Example: Postfix Evaluation

Unit Tests

- *After lecture, the solution will be pasted into the slide here.*
- I ended up skipping this during lecture.

Stack

Programming Assignment #2: Undo/Redo History

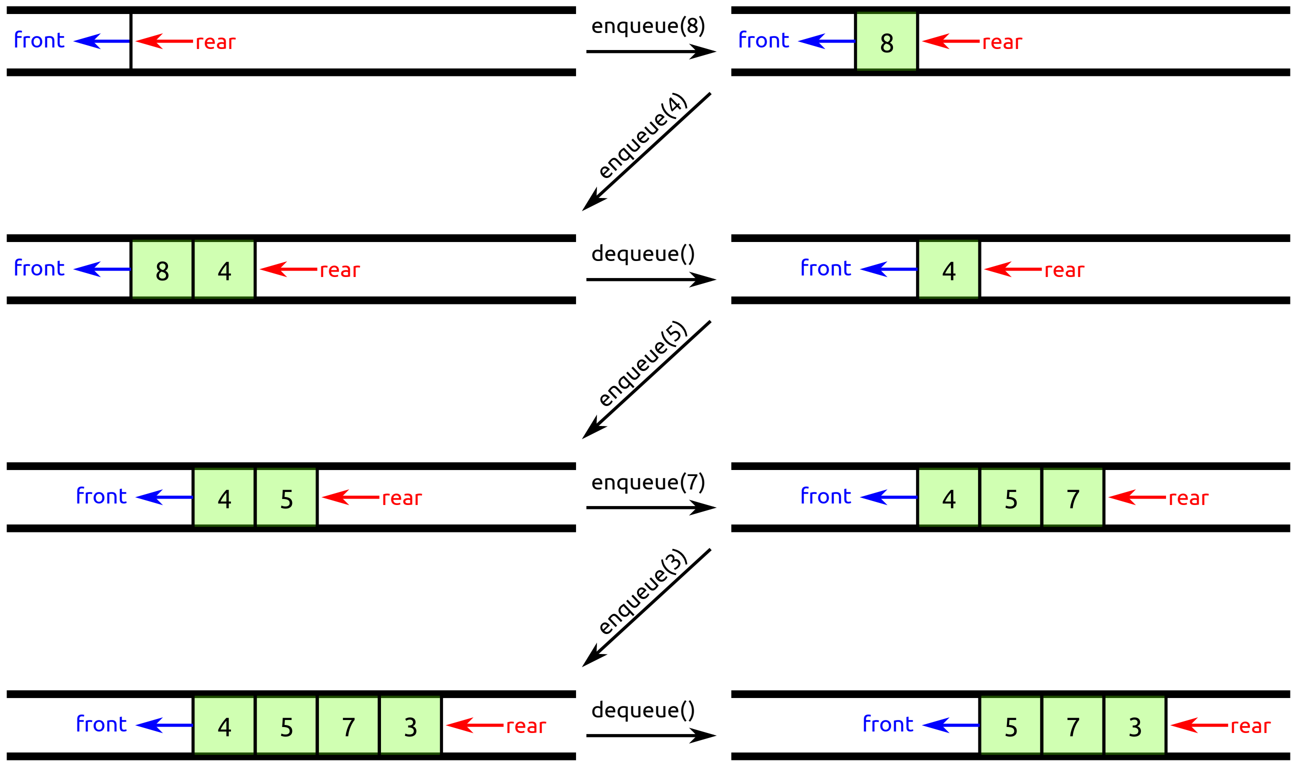
- *At this point in the lecture, I talked about the undo/redo history on the document camera.*

Queue

Overview

- Elements are added at one end and removed at the other.
- FIFO, first-in first-out.

Example #1



- Just to be clear (since I mixed it up a bit during lecture):
  - In above illustration, insertion occurs at the "rear"; removal, at the "front".
  - "front" and "rear" in the above illustration correspond to the back of the underlying list (see next slide) and front of it, respectively.
  - The exact terms don't matter; what matters is insertion and removal occur at opposite ends.

Queue

Textbook's Implementation[1]

```
class Queue:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def enqueue(self, item):
        self.items.insert(0,item)

    def dequeue(self):
        return self.items.pop()

    def size(self):
        return len(self.items)
```

queue.py

- "dequeue" is pronounced "dee-queue".
- Do each of these methods take constant time?
    - No. All of them except `enqueue()` do. `enqueue()` takes linear time; need to slide the rest of the list back.

1. From section 4.12 of *Problem Solving with Algorithms and Data Structures using Python* by Brad Miller and David Ranum.

Deque

- Double-ended queue.
- Items can be added/removed at either end.
  - Neither LIFO nor FIFO.
- Prounounced "deck".

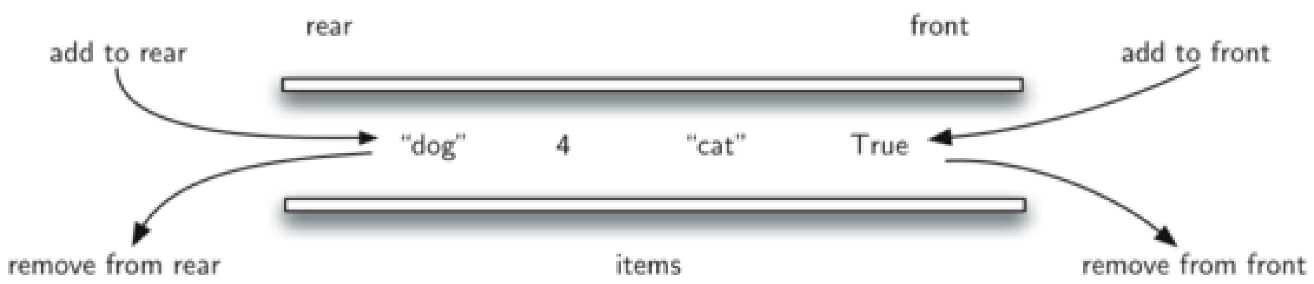


Image from section 4.15 of *Problem Solving with Algorithms and Data Structures using Python* by Brad Miller and David Ranum.

Deque

Textbook's Implementation[1]

```python
class Deque:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def addFront(self, item):
        self.items.append(item)

    def addRear(self, item):
        self.items.insert(0,item)

    def removeFront(self):
        return self.items.pop()

    def removeRear(self):
        return self.items.pop(0)

    def size(self):
        return len(self.items)
```

deque.py

- Rear is front of list.
- `addRear()`/`removeRear()` take $\Theta(n)$ in the worst case.
- Other methods take constant time[2].

1. From section 4.17 of *Problem Solving with Algorithms and Data Structures using Python* by Brad Miller and David Ranum.

2. As stated during the 08/19 lecture and as supported here, since `append()`, in the worst case, takes linear time *but* amortized constant time, we should say `addFront()` takes amortized constant time, not constant time (which is technically incorrect), but you don't need to worry about this.

Abstract Data Type (ADT)

Definition

- Set of objects plus a set of operations.
- **Says nothing about how those operations are implemented.**
  - Different implementations → different runtimes (usually).

Examples (we've seen)

- Queue.
- Stack.
- Deque.

Nonexamples

- Array (i.e. Python list).
- Linked list (coming soon).
- Binary heap.
- Hash tables.

Examples (we'll see)

- Unordered and ordered lists.
- Set.
- Map.
- Tree.
- Graph.

Unordered List

Unordered List ADT

- "Collection of items where each item holds a relative position with respect to the others."[1]
- A Python list (called an "array"/"vector" in other languages) is a way of implementing the ADT, but there are other ways to do so.
  - Particularly, Python lists store data contiguously in memory (hence constant time indexing), but the List ADT doesn't require this.

Some Possible Operations

- `add(item)`
- `remove(item)`
- `search(item)` (boolean function)
- `isEmpty()`
- `size()`
- `append(item)`
- `index(item)`
- `insert(pos,item)`
- `pop()`
- `pop(pos)`

1. Source: section 4.17 of *Problem Solving with Algorithms and Data Structures using Python* by Brad Miller and David Ranum.

Linked List

Overview

- Maintains ordering of elements without guaranteed contiguous placement in memory.

Figure 1: Items Not Constrained in Their Physical Placement

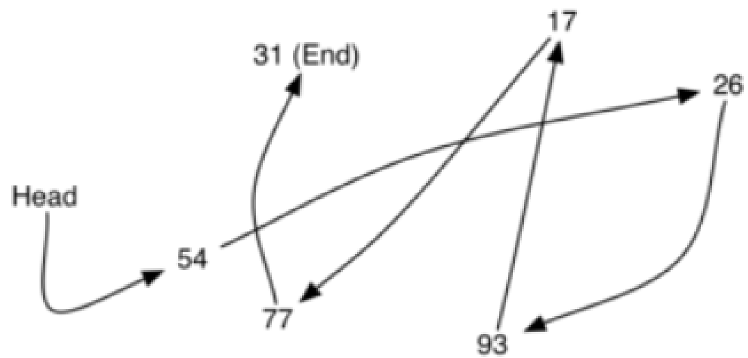


Figure 2: Relative Positions Maintained by Explicit Links.

- Only need to know where the head is.

Linked List

Node

Overview

- Linked list is chain of nodes.
- Each node has:
    1. Data.
    2. Reference to next node. (singly linked list)
        - Doubly linked list: reference to previous node as well.
        - Last node's reference is None. ("grounding the node")

Textbook's Implementation

```python
class Node:
    def __init__(self,initdata):
        self.data = initdata
        self.next = None

    def getData(self):
        return self.data

    def getNext(self):
        return self.next

    def setData(self,newdata):
        self.data = newdata

    def setNext(self,newnext):
        self.next = newnext
```

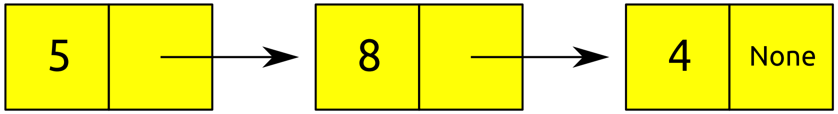
- Uses getters/setters. (doesn't need to)

Linked List

Node

Example

```
n1 = Node(5)
n2 = Node(8)
n3 = Node(4)
n1.setNext(n2)
n2.setNext(n3)
```



- Only need head to have access to entire list.
- Could also do:

```
head = Node(5)
head.setNext(Node(8))
head.getNext().setNext(Node(4))
```

Textbook's Implementation

```
class Node:
    def __init__(self,initdata):
        self.data = initdata
        self.next = None

    def getData(self):
        return self.data

    def getNext(self):
        return self.next

    def setData(self,newdata):
        self.data = newdata

    def setNext(self,newnext):
        self.next = newnext
```

Unordered List

- Reminder: unordered list ADT can be implemented in many ways.

Array/Vector[1] Implementation

- Use Python list (trivial).

1. The C/C++ terms.

Unordered List

Linked List Implementation (Textbook's)

Starting Off

```
class UnorderedList:

    def __init__(self):
        self.head = None

    def isEmpty(self):
        return self.head == None

    ...
```

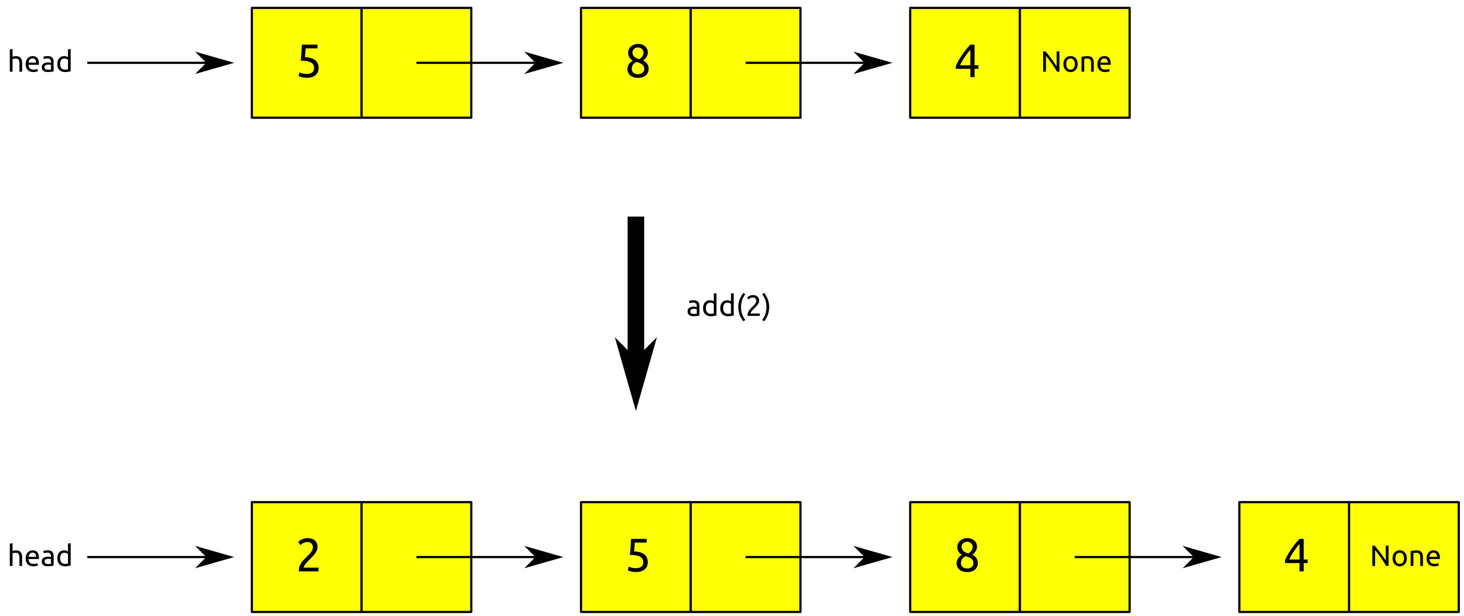
Unordered List

Linked List Implementation

`add(item)`[1]

- Insert element at front of list.
- Constant time.

```
def add(self,item):
    temp = Node(item)
    temp.setNext(self.head)
    self.head = temp
```



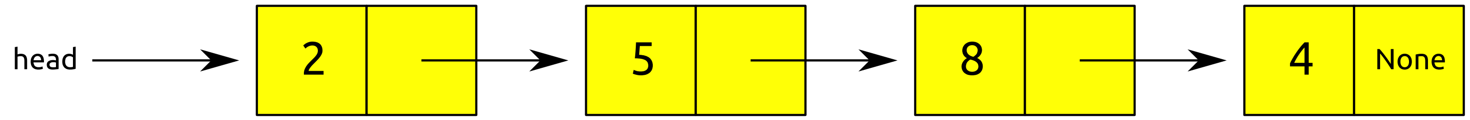
1. I won't indent these, but of course, they should be indented into the class definition.

Unordered List

Linked List Implementation

`size()` and `search(item)`

```
def size(self):
    current = self.head
    count = 0
    while current != None:
        count = count + 1
        current = current.getNext()

    return count

def search(self,item):
    current = self.head
    found = False
    while current != None and not found:
        if current.getData() == item:
            found = True
        else:
            current = current.getNext()

    return found
```

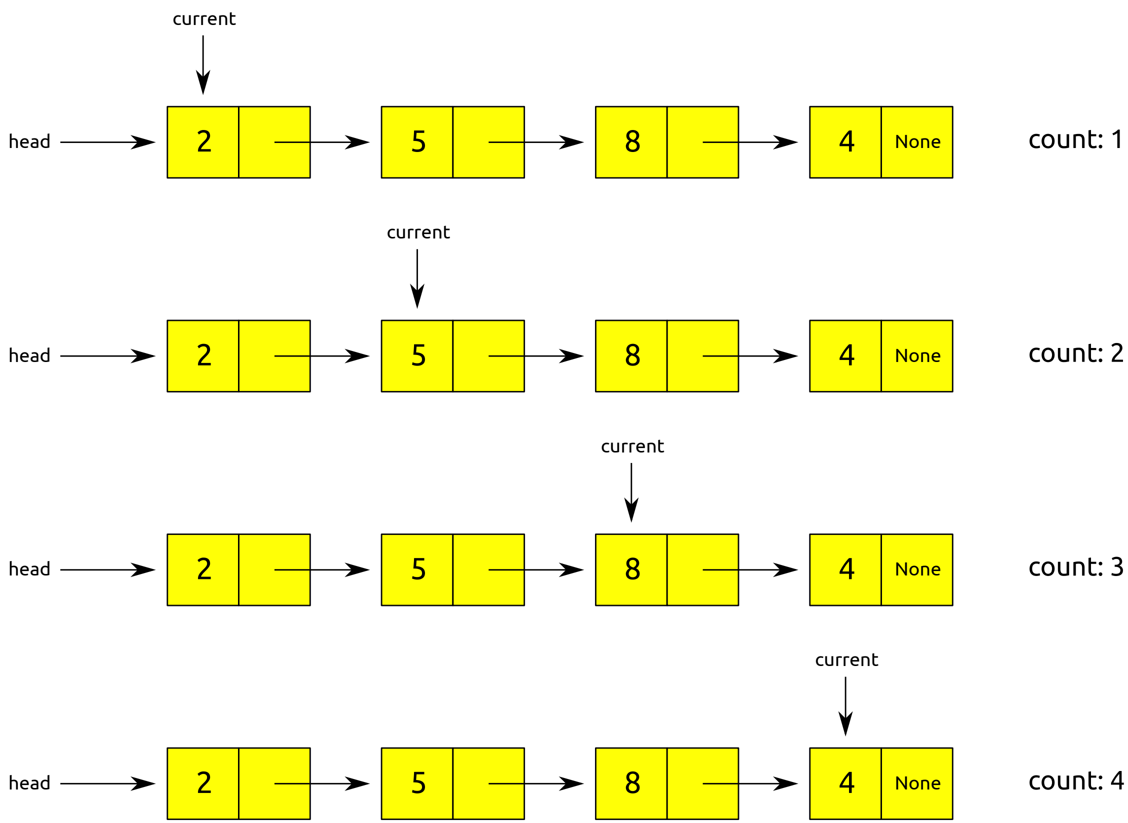


Unordered List

Linked List Implementation

`size()`: Step-by-Step

```
def size(self):
    current = self.head
    count = 0
    while current != None:
        count = count + 1
        current = current.getNext()

    return count
```

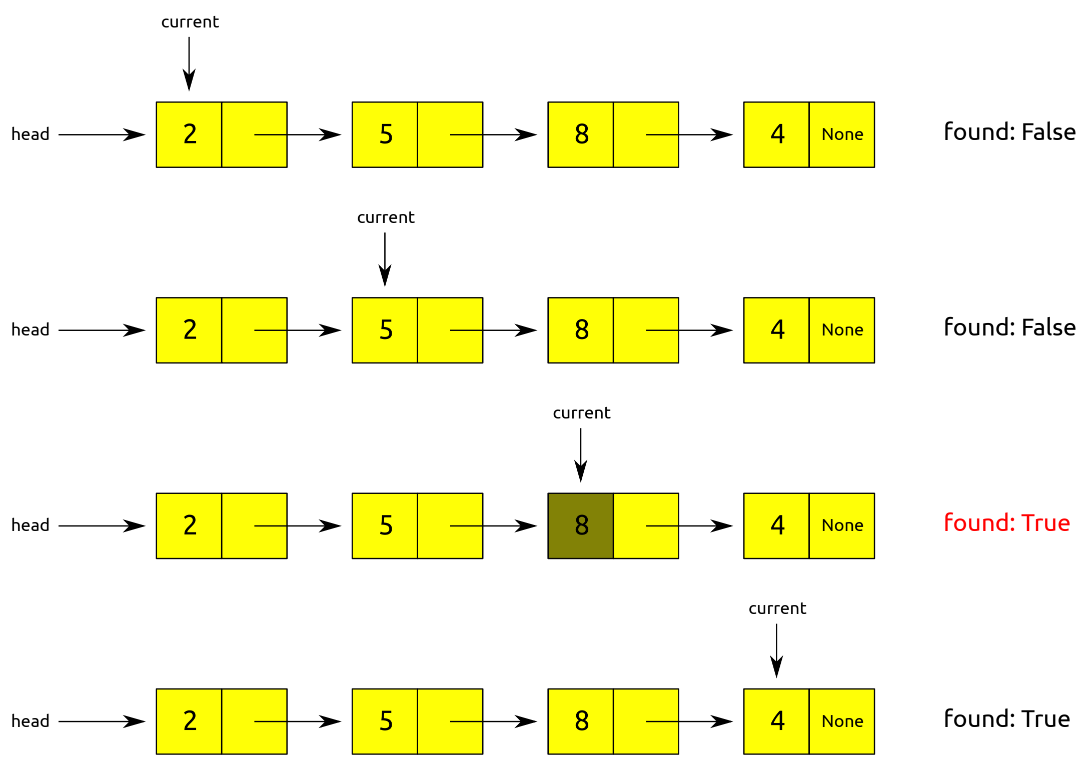


Unordered List

Linked List Implementation

`search(8)`: Step-by-Step

```
def search(self,item):
    current = self.head
    found = False
    while current != None and not found:
        if current.getData() == item:
            found = True
        else:
            current = current.getNext()
    return found
```



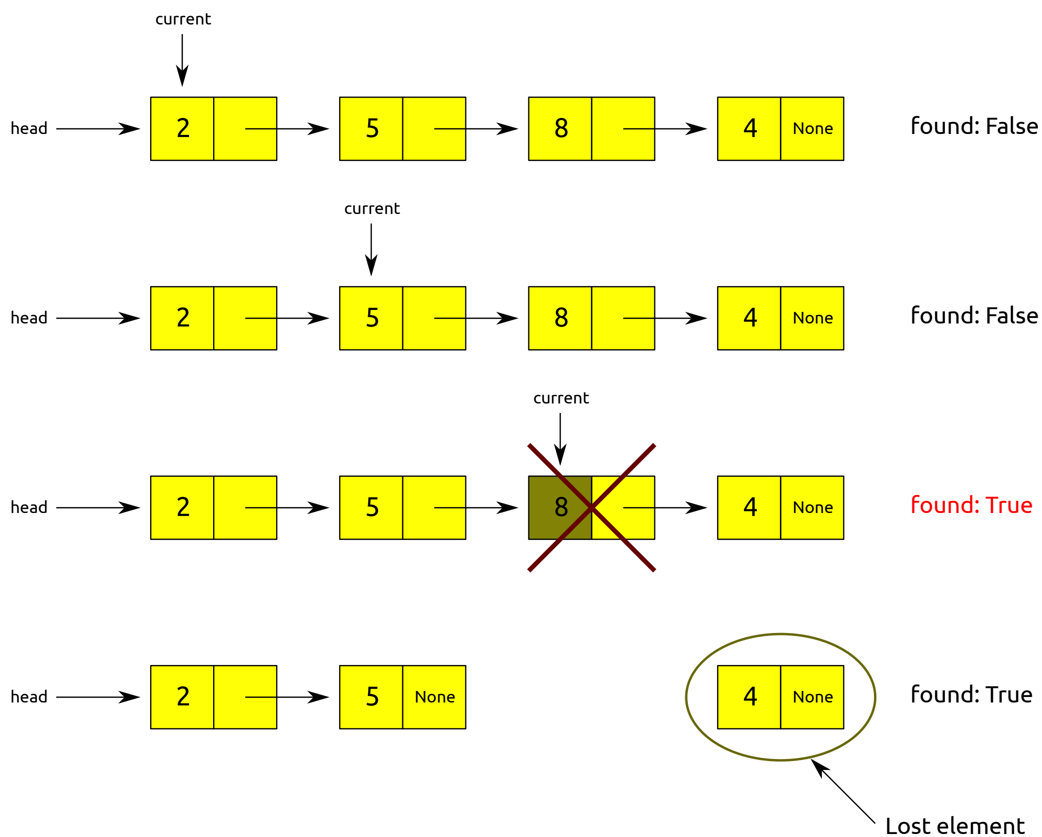
Unordered List

Linked List Implementation

`remove(item)`: Brainstorming

- Traverse list, element-by-element (like `size()` and `search()`), until find `item`.
- Remove it.
- *Issue*: The `next` link of the node before it will be broken ⇒ lose access to list's remainder.

Example: `remove(8)`



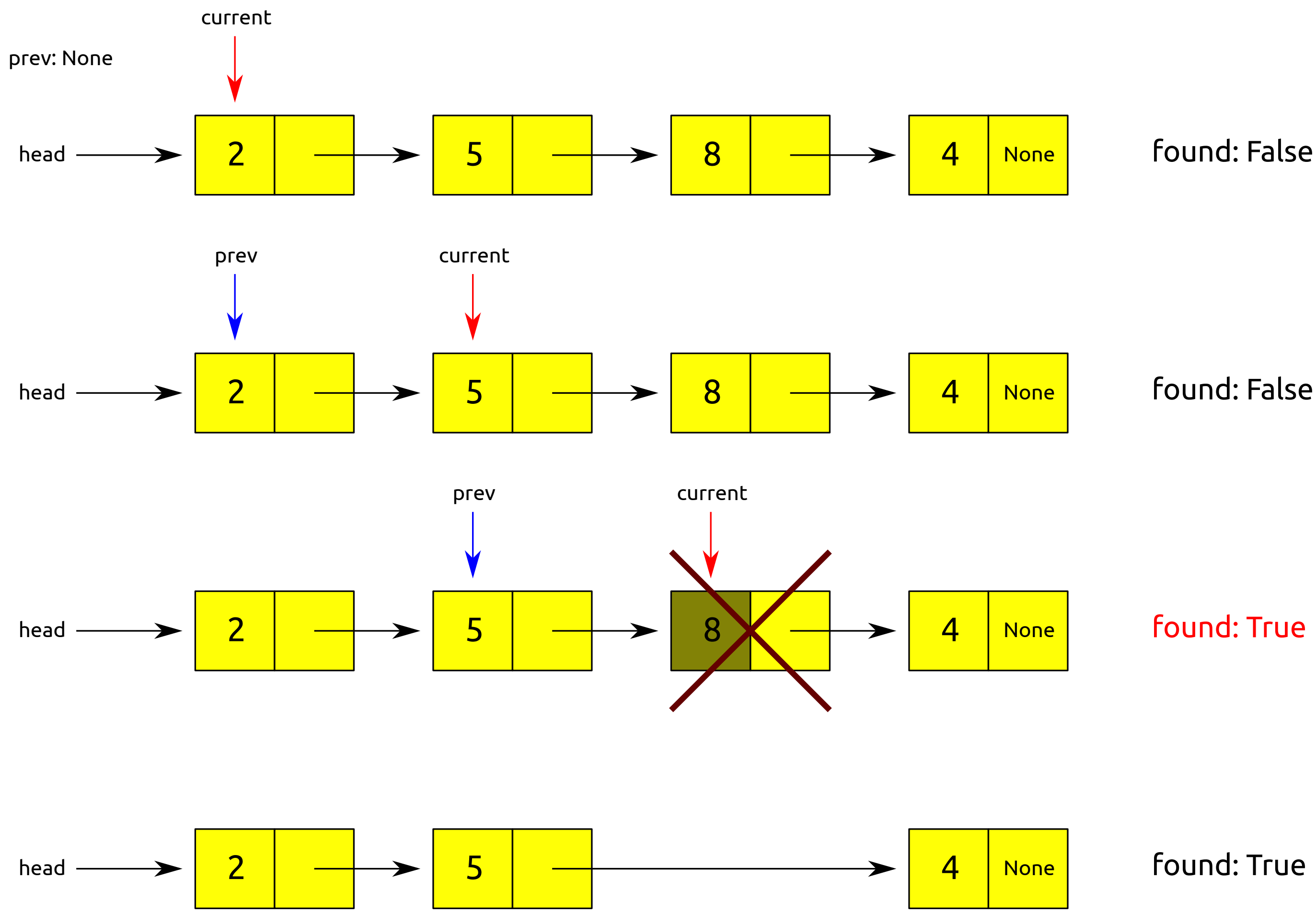
- Need to keep track of what the node before the removed one is, so can set its `next` reference.

Unordered List

Linked List Implementation

`remove(item)`: What We Want

- Once find `item`, adjust `next` member of `prev` node.



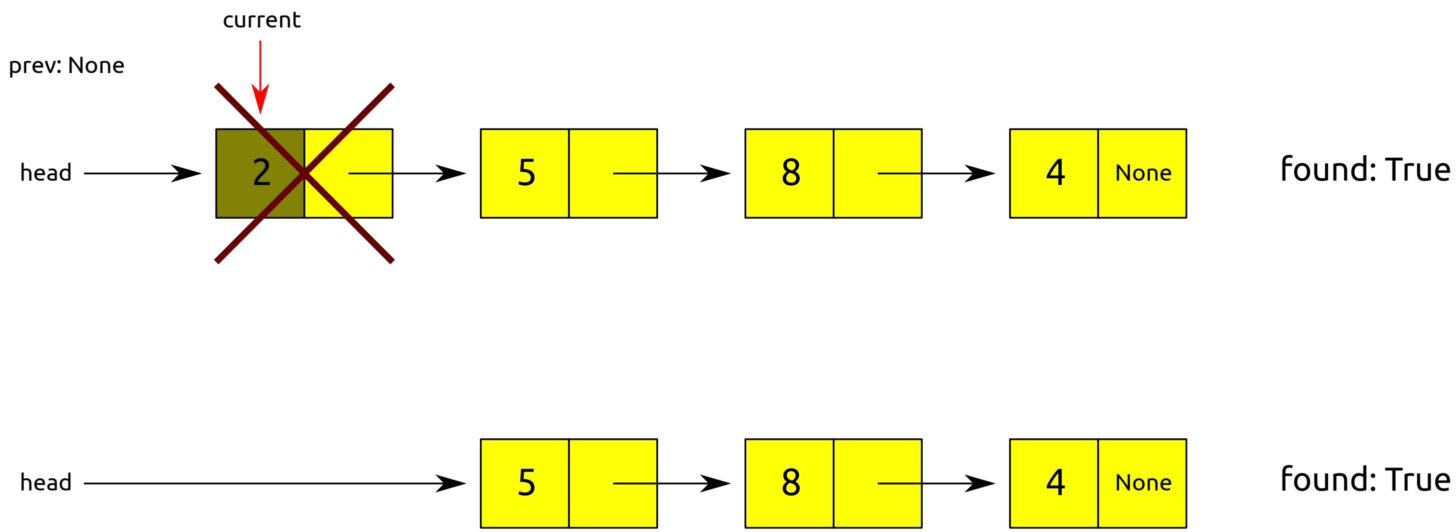
- Issue: what if `item` is at head?

Unordered List

Linked List Implementation

`remove(item)`: Dealing with Head Case

- If find item at start of list, adjust `head`.



Unordered List

Linked List Implementation

`remove(item)`: Code

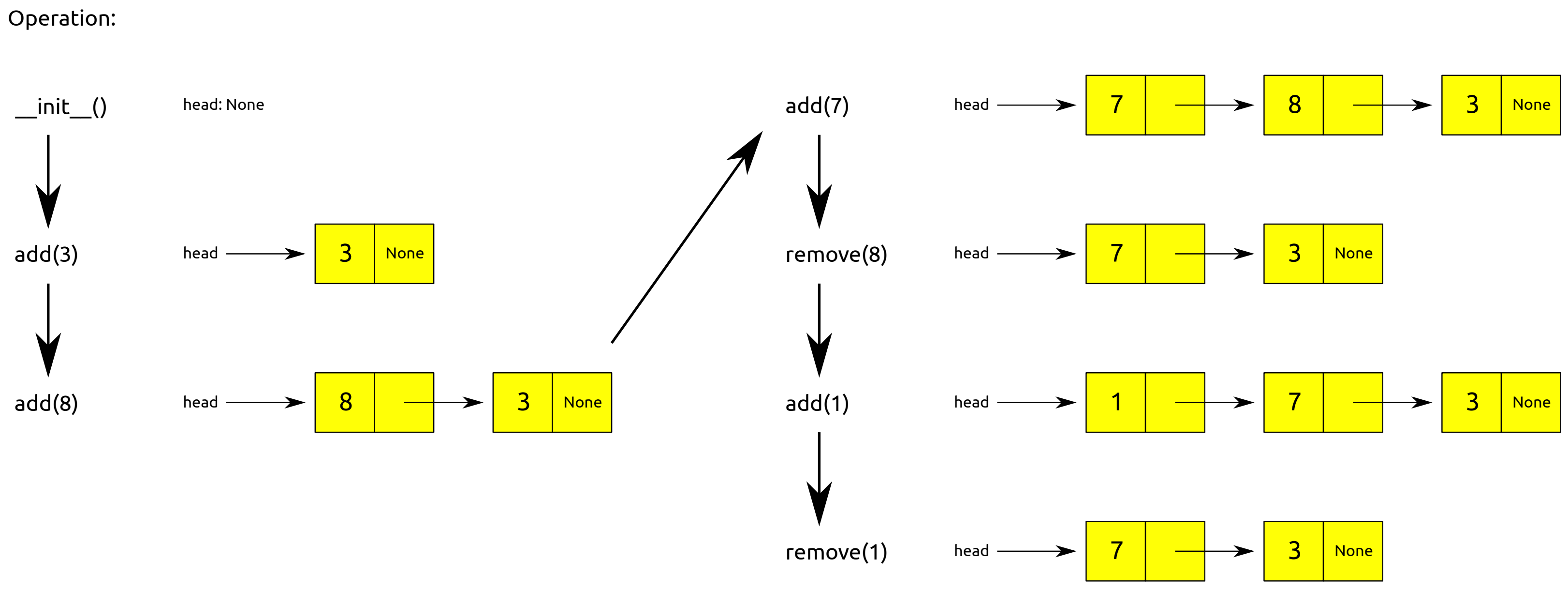
```
def remove(self,item):
    current = self.head
    previous = None
    found = False
    while not found:
        if current.getData() == item:
            found = True
        else:
            previous = current
            current = current.getNext()

    if previous == None:
        self.head = current.getNext()
    else:
        previous.setNext(current.getNext())
```

- Assumes item is in the list.
- Worst-case time complexity: $\Theta(n)$.
    - Constant time work once find `item`.
- Removed node is automatically destroyed once `remove()` ends, due to no longer being referenced.

Unordered List

Linked List Implementation

Final Example



Unordered List

Linked List Implementation

```
class Node:
    def __init__(self,initdata):
        self.data = initdata
        self.next = None

    def getData(self):
        return self.data

    def getNext(self):
        return self.next

    def setData(self,newdata):
        self.data = newdata

    def setNext(self,newnext):
        self.next = newnext


class UnorderedList:
    def __init__(self):
        self.head = None

    def isEmpty(self):
        return self.head == None

    def add(self,item):
        temp = Node(item)
        temp.setNext(self.head)
        self.head = temp

    def size(self):
        current = self.head
        count = 0
```

```
        while current != None:
            count = count + 1
            current = current.getNext()

        return count

    def search(self,item):
        current = self.head
        found = False
        while current != None and not found:
            if current.getData() == item:
                found = True
            else:
                current = current.getNext()

        return found

    def remove(self,item):
        current = self.head
        previous = None
        found = False
        while not found:
            if current.getData() == item:
                found = True
            else:
                previous = current
                current = current.getNext()

        if previous == None:
            self.head = current.getNext()
        else:
            previous.setNext(current.getNext())
```

Unordered List

Comparison of Implementations (Worst-Case Time Complexity)

Operation	**Python List**	**Linked List**
Indexing ("random access")	$\Theta(1)$	$\Theta(n)$
`add(item)`	$\Theta(n)$ (insert front), amortized $\Theta(1)$ (insert back)	$\Theta(1)$ (insert front)
`remove(item)`	$\Theta(n)$ (regardless of location of `item`)	$\Theta(n)$ (at back)
`search(item)`	$\Theta(n)$	$\Theta(n)$
`size()`	$\Theta(1)$	$\Theta(n)$ (naive), $\Theta(1)$ (smart)

Unordered List

Improving Linked List Implementation

Constant Time `size()` Implementation

- Create a member variable to keep track of size[1].
- Update when appropriate (when add/remove).
- Changes:

```python
class UnorderedList:
    def __init__(self):
        self.head = None
        self.num_elems = 0

    ...

    def add(self,item):
        temp = Node(item)
        temp.setNext(self.head)
        self.head = temp
        self.num_elems += 1

    def size(self):
        return self.num_elems
```

```python
    def remove(self,item):
        current = self.head
        previous = None
        found = False
        while not found:
            if current.getData() == item:
                found = True
            else:
                previous = current
                current = current.getNext()

        if previous == None:
            self.head = current.getNext()
        else:
            previous.setNext(current.getNext())
        self.num_elems -= 1
```

1. This is what a Python list does behind-the-scenes.

Unordered List

Improving Linked List Implementation

Implement `append(item)`

- Appends item to back of list.
- So that we can even tell the difference vs. `add()`, let's add `print_all()`.

```
class UnorderedList:
    ...
    def print_all(self):
        current = self.head
        while current != None:
            print(current.getData(), end=' ')
            current = current.getNext()
        print()

    def append(self,item):
        pass

mylist = UnorderedList()
mylist.add(31)
mylist.add(77)
mylist.add(26)
mylist.add(54)
mylist.print_all()
```

```
54 26 77 31
```

Unordered List

Improving Linked List Implementation

Implement `append(item)`

- Two scenarios:
    1. List is nonempty: keep iterating until `previous` references last node.
    2. List is empty: set new node as head.

```
class UnorderedList:
    ...
    def append(self,item):
        current = self.head
        previous = None
        # Find end of list.
        while current != None:
            previous = current
            current = current.getNext()

        temp = Node(item)
        if previous == None:
            self.head = temp
        else:
            previous.setNext(temp)
```

```
31 15
54 26 77 31 15
54 26 77 31 15 42
```

```
mylist = UnorderedList()
mylist.append(15)
mylist.add(31)
mylist.print_all()
mylist.add(77)
mylist.add(26)
mylist.add(54)
mylist.print_all()
mylist.append(42)
mylist.print_all()
```

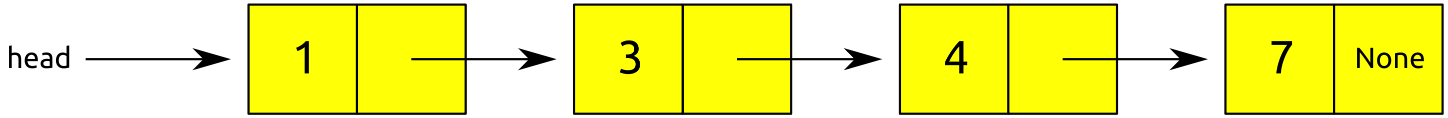
- Takes linear time... can improve.

Ordered List

- Maintain the list's values in a sorted order.

Linked List Implementation

- Two ways of maintaining the sorted order:
    1. Whenever an element is inserted, sort the entire list.
    2. Insert new element into correct position to maintain sort.
- Given that sorting algorithms take $O(n \lg n)$ or $O(n^2)$ time, let's not do #1.
- Compared to `UnorderedList`:
    - Modify `search()` with small speedup.
    - Need change `add()`.
    - `size()` and `remove()` stay same.
    - `append()` wouldn't make sense.



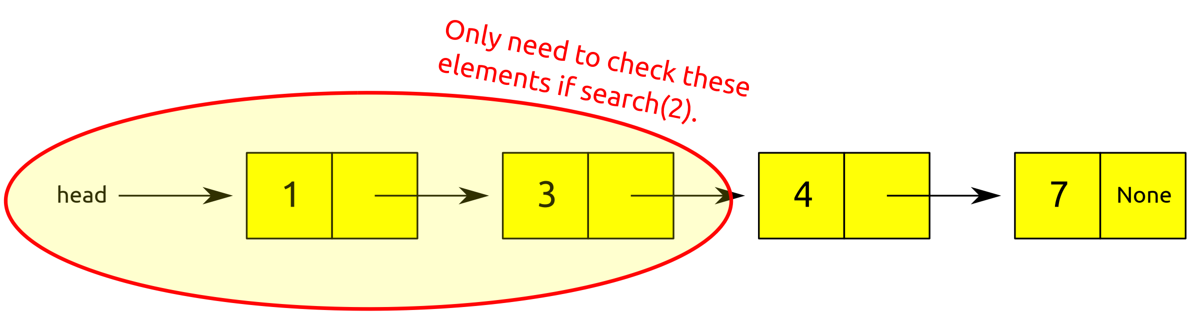
Ordered List

Linked List Implementation

`search(item)`

- Stop early if don't find `item`.

```
def search(self,item):
    current = self.head
    found = False
    stop = False
    while current != None and not found and not stop:
        if current.getData() == item:
            found = True
        else:
            if current.getData() > item:
                stop = True
            else:
                current = current.getNext()

    return found
```

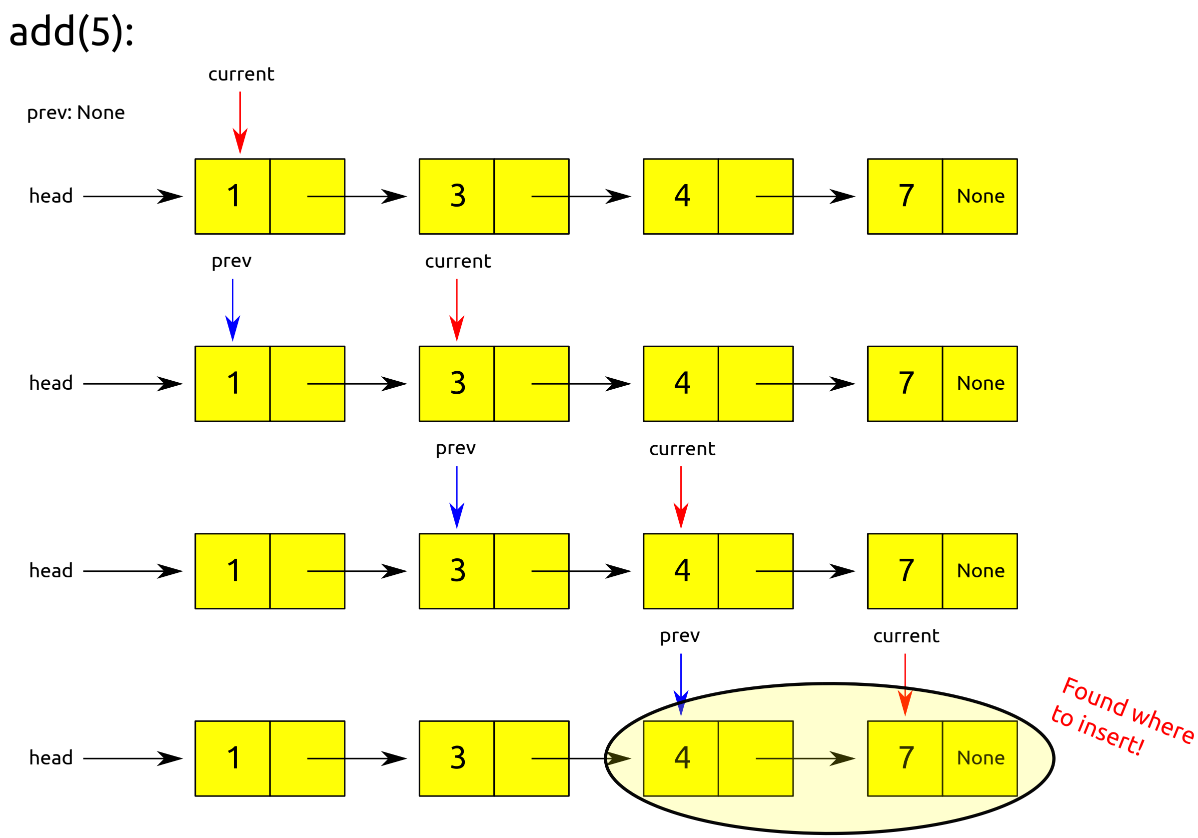


Ordered List

Linked List Implementation

`add(item)`: Brainstorming

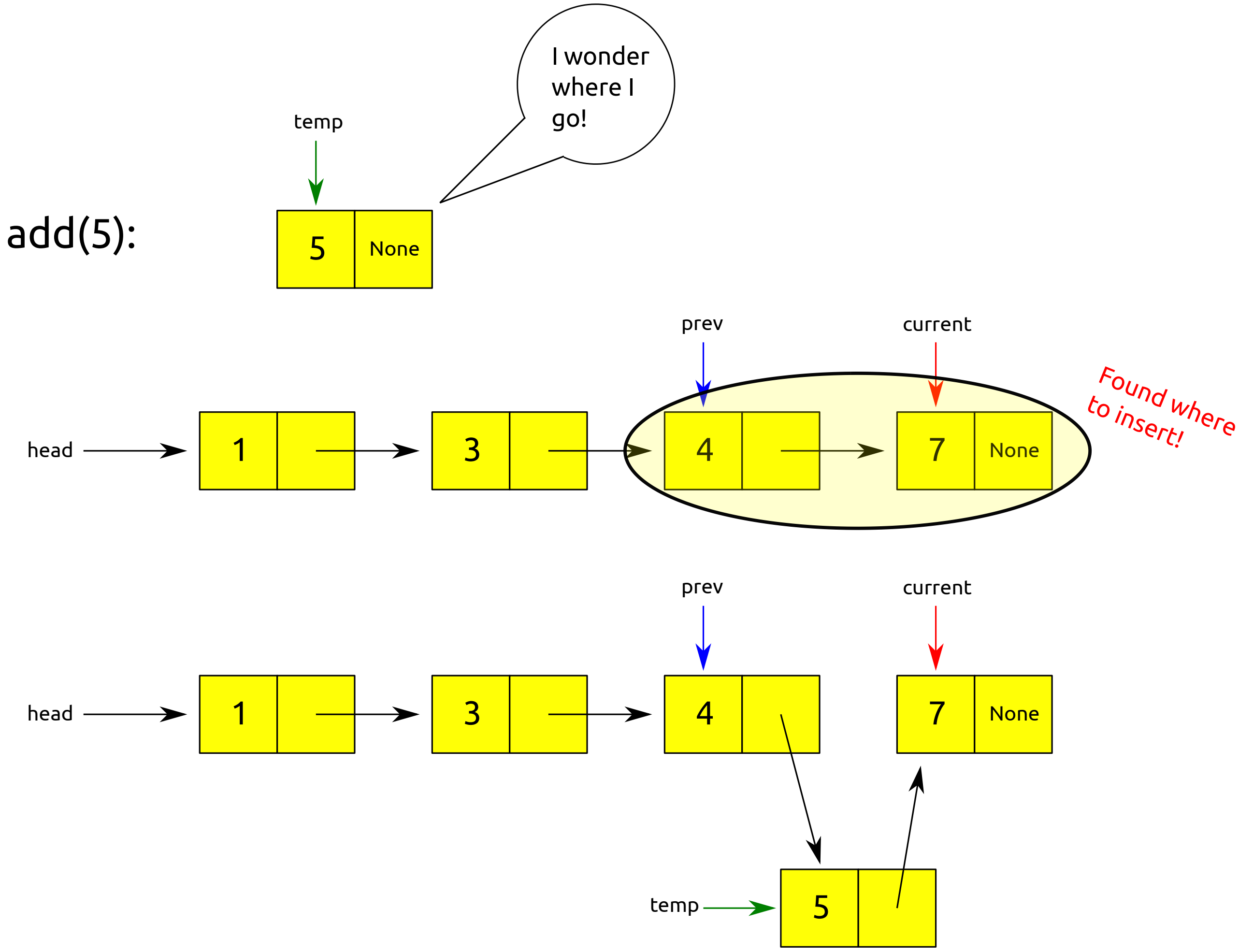
- Adding to front would violate sorted order.
- Steps:
  1. Find where to insert `item`.
  2. Create new node and set its `next` member to node right after the insertion spot.
  3. Update `next` member of node right before the insertion spot.



Ordered List

Linked List Implementation

`add(item)`: Brainstorming



Ordered List

Linked List Implementation

`add(item)`: Implementing

```
def add(self,item):
    current = self.head
    previous = None
    stop = False
    while current != None and not stop:
        if current.getData() > item:
            stop = True
        else:
            previous = current
            current = current.getNext()

    temp = Node(item)
    if previous == None:
        temp.setNext(self.head)
        self.head = temp
    else:
        temp.setNext(current)
        previous.setNext(temp)
```

1. If doing version with `num_elems` member, then `add()` would still increment it. The above version is the book's.

References / Further Reading

- Chapter 4 of *Problem Solving with Algorithms and Data Structures using Python* by Brad Miller and David Ranum.
  - The author of this book created a Python module containing "all of the common data structures and implementations of some algorithms as presented in the book". You can find it here. At least you are running Python from the command line / terminal, you would have to install it by using a Python installation tool called pip. I am unsure how to do it through Python IDLE, the Mu editor, or PyCharm, but you can look it up.
- *Introduction to Algorithms* by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein (Third Edition).
  - Chapter 10: some linear data structures.
  - Chapter 17: amortized analysis.
- Webpage containing the amortized worst-case time complexities of various list methods (as talked about during 08/19 lecture): here.