(06) Linear Data Structures Part 2: Stacks

Video (23 mins): https://youtu.be/8DsBibOIsB8

So the Last shall be First, and the First Last

Linear data structures

- Stack
- Queue
- Priority Queue

Reference

- You will need the supporting Python file for the tutorial exercises.
- Download **LinearDSLab.py** from eLearn
- Instructions:
  - Import all the linear data structures from **LinearDSLab**
  - Need to do this step each time you open a new terminal

or **`python3`** for MacOS

```
python
>>> from LinearDSLab import *
```

Linear Data Structures

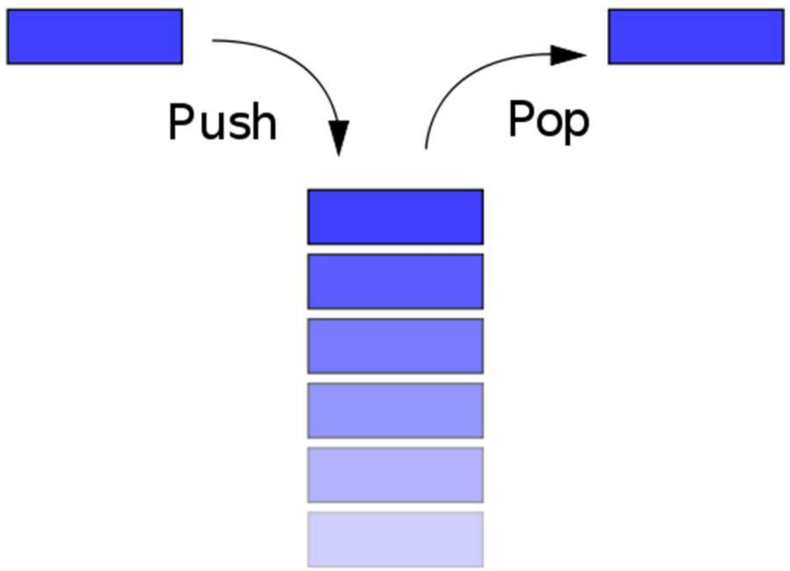
- Only ONE data element is accessible at any point of time
- Simplifies programming logic:
  - Don't need to keep track of indices.
- The key question is **which** element is accessible.
- Three types:
  - Stack
  - Queue
  - Priority Queue

queue

stack

Stack

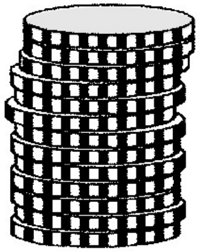
- Stack is a data structure with LIFO (Last In, First Out) property: The last item placed on the stack will be the first item removed.
- Defined primarily by three main operations: `push`, `pop`, `peek`

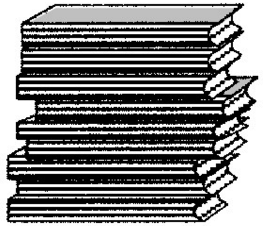


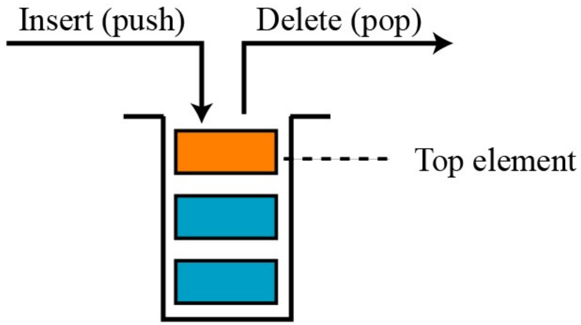
http://en.wikipedia.org/wiki/Stack_(abstract_data_type)

Stacks – Motivating Examples


Tower of Hanoi


Stack of coins


Stack of books



Computer stack

- Computer Applications of Stacks
  - Keyboard erase/backspace key
  - Supporting UNDO/BACK operation
  - Supporting Recursion and Backtracking

Backspace Key

Bottom of stack

Top of stack

“The quick brown fox jumps over the lazy dog”

“The quick brown fox jumps over the lazy do”

Application: Function Call Stack

```
def f1(n):
   return 2 * f2(n)
```

```
def f2(n):
   return 2 * f3(n)
```

```
def f3(n):
   return 2 * f4(n)
```

```
def f4(n)
   return n
```

Call Stack

f4	Top of stack
f3	
f2	
f1	Bottom of stack

See example:
http://cs.nyu.edu/courses/spring07/V22.0101-002/19slide.ppt
(slides 12-22)

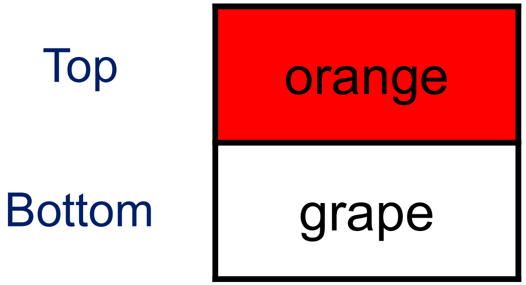
Stack Overflow?

- No, not the website where you find answers for technical questions.
- Try this:

```
def stack_overflow(a):
…   print(a)
…   stack_overflow(a+1)
…
>>> stack_overflow(1)
???
```

Stack Operation: `push`

- Places a new data element to the top of a stack s



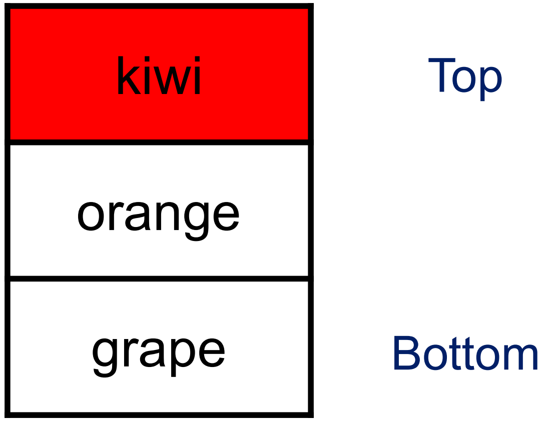
Before

Create stack (Before)

```
>>> s = Stack()
>>> s.push("grape")
>>> s.push("orange")
>>> s.display()
```

Place new data element on top of stack (After)

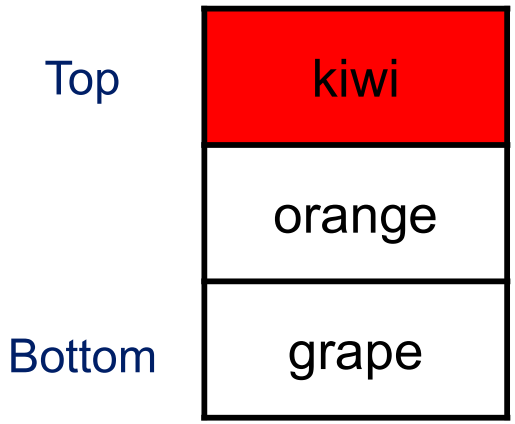
```
>>> s.push("kiwi")
```



After

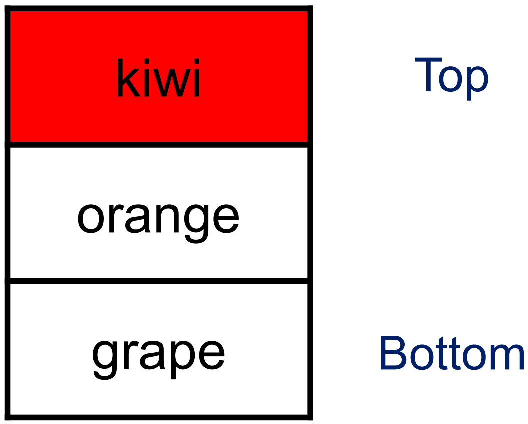
Stack Operation: `peek`

- Inspects the data element on top of the stack without removing it



Before

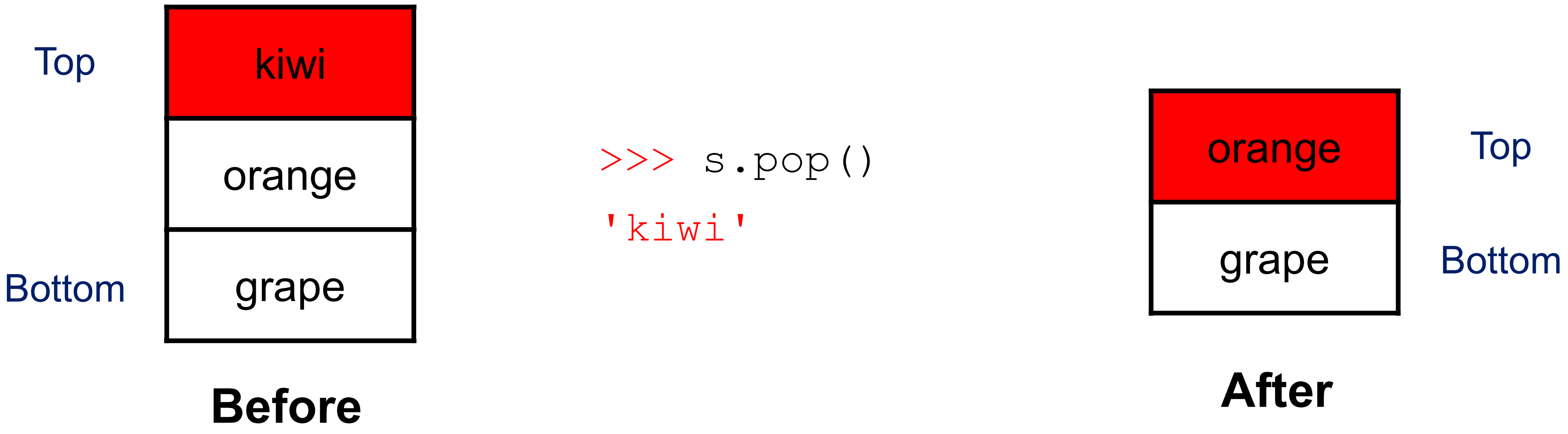
```
>> s.peek()
'kiwi'
```



After

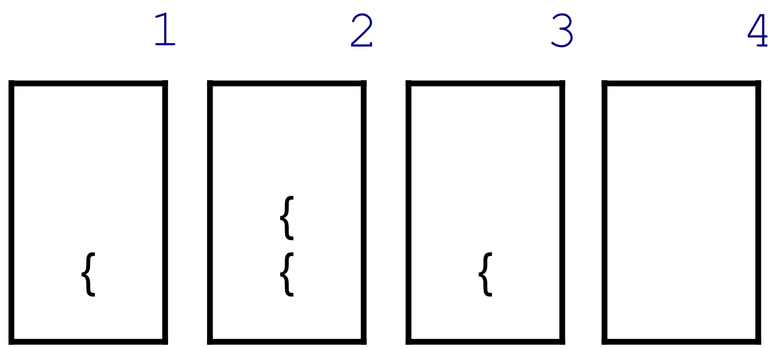
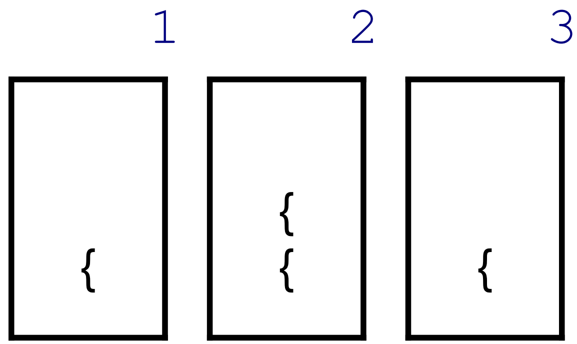
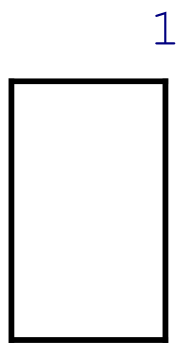
Stack Operation: `pop`

- Removes and retrieves the data element on top of the stack.



Example: Check Balancing Braces

- Parentheses are commonly used in mathematical operations.
  - e.g., (a + b) x (c + d)
- Braces are commonly used in programming languages (e.g., Java).
  - e.g., `if(condition is true) { #execute statement }`
- Imbalanced braces may cause errors.
- Braces are balanced if:
  - there is a matching closing brace for each opening brace;
  - we do not put a closing brace before an opening brace.
- How do we detect balanced braces?
  - Push “{” into stack
  - Pop from stack when “}” is encountered

Input	Stack states	Stack operations
a{b{c}d}		1. push("{") 2. push("{") 3. pop() 4. pop() Stack empty: *balanced*
ab{{c}d		1. push("{") 2. push("{") 3. pop() Stack not empty: *not balanced*
a}b{cd		1. pop() Stack empty when "}" is encountered: *not balanced*

```
# checks whether text contains balanced braces using stack
def check_braces(text):   # function defined in LinearDSLab.py
  s = Stack()
  for ch in text:
    if ch == "{":
      s.push(ch)           # push opening brace into stack
    elif ch == "}"
      if s.count() > 0:    # ensure stack not empty when pop
        s.pop()            # pop when closing brace encountered
      else:
        return False       # returns False since stack is empty
  return s.count() == 0    # returns True if stack is empty

# test cases
>>> check_braces("a{b{c}d}") # returns True
>>> check_braces("ab{{c}d")  # returns False
>>> check_braces("a}b{cd")   # returns False
```

List-based Implementation of push

Use a list named li to contain data elements.
First element (index 0) is top of the stack.

```
def push(li, item):
    li.insert(0,item)
```

0	orange
1	grape

Before

```
>>> li = ["orange", "grape"]
>>> push(li, "kiwi")
```

0	kiwi
1	orange
2	grape

After

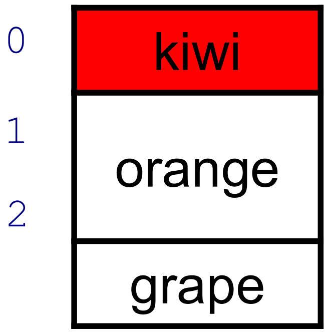
shift

Complexity?

`List`-based Implementation of `peek`

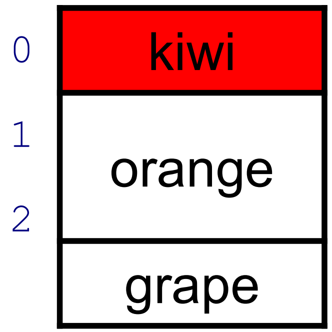
Use a list named `li` to contain data elements.
First element (index 0) is top of the stack.

```
def peek(li):
    if len(li) > 0:
        return li[0]
```



Before

```
>>> peek(li)
'kiwi'
```



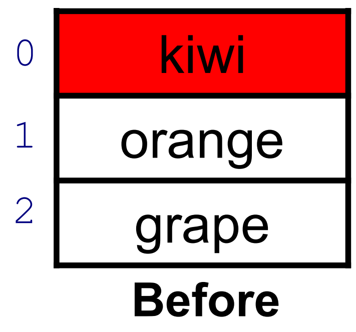
After

Complexity?

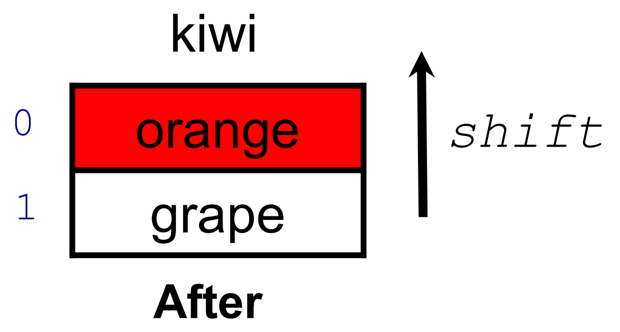
`List`-based Implementation of `pop`

Use a list named `li` to contain data elements.
First element (index 0) is top of the stack.

```
def pop(li):
    if len(li) > 0:
        item = li[0]
        del li[0]
        return item
```




```
>>> pop(li)
'kiwi'
```



Complexity?

Stack and Recursion

- Stacks can be used to create a non-recursive version of a recursion.
- When a recursive algorithm is compiled, it is typically "reimplemented" as a stack-based iterative algorithm by the compiler.
- How is it done?
  - Create a new stack.
  - Push initial parameters onto a stack.
  - Iterate till stack is empty:
    - pop parameter from stack
    - if <u>base case</u>:
      do not push any more parameter to the stack
    - else (<u>reduction step</u>):
      push onto the stack the parameters that'd have been used in recursion
      note to push earlier function call later

Example: Fibonacci series

- Fibonacci series: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
- Each number is the sum of the previous two numbers
  - `fibonacci(0) = 0`
  - `fibonacci(1) = 1`
  - `fibonacci(2) = 1 + 0 = 1`
  - `fibonacci(3) = 1 + 1 = 2`
  - `fibonacci(4) = 2 + 1 = 3`
  - `fibonacci(5) = 3 + 2 = 5`
  - `...`
- **Reduction**: `fibonacci(n) = fibonacci(n-1)+fibonacci(n-2)`
- **Base cases**: `fibonacci(0) = 0, fibonacci(1) = 1`

Recursive Version of Fibonacci

```
def fibonacci(n):
    print(n)          #to show value at each step
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

```
>>> num = fibonacci(4)
4
3
2
1
0
1
2
1
0
>>> num
3
```

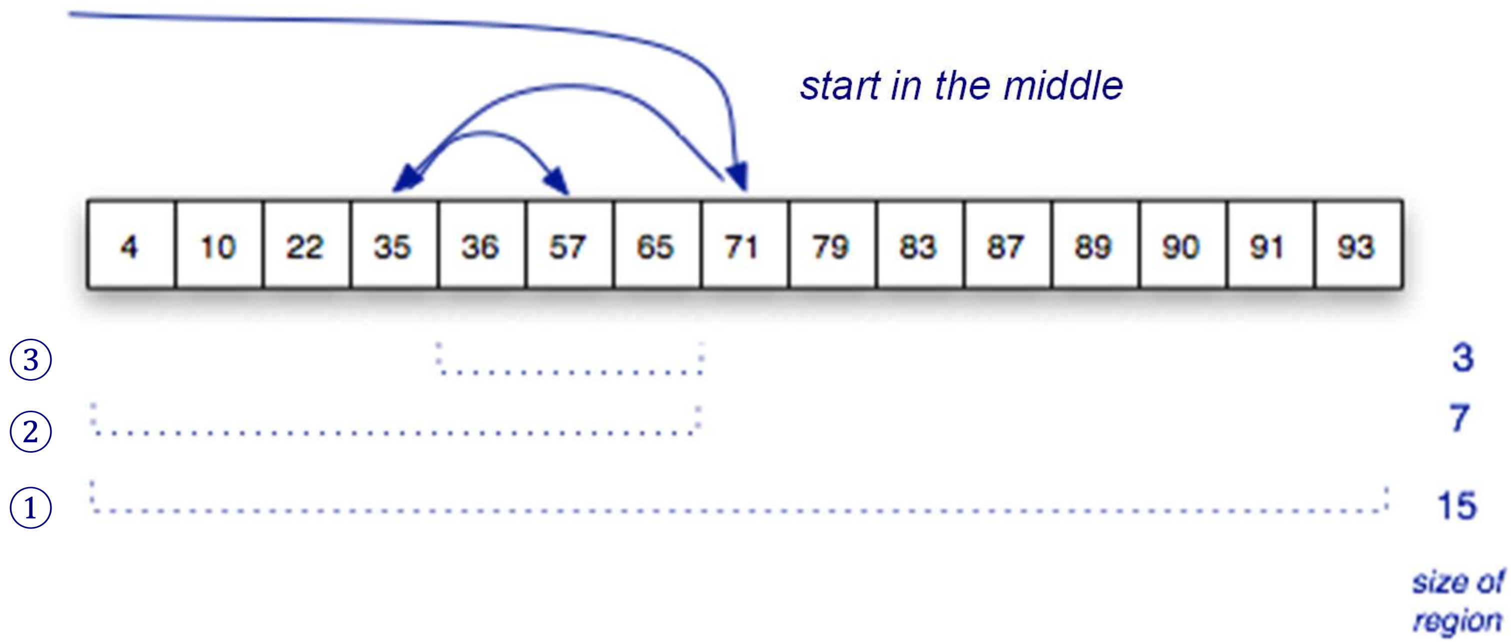
Non-recursive Fibonacci with Stack

```python
def fibonacci_stack(n):
    s = Stack()
    s.push(n)
    result = 0
    while s.count() > 0:
        s.display()
        current = s.pop()
        if current == 0:
            result += 0
        elif current == 1:
            result += 1
        else:
            s.push(current - 2)
            s.push(current - 1)

    return result
```

```
>>> num = fibonacci_stack(4)
[4] <= top
[2, 3] <= top
[2, 1, 2] <= top
[2, 1, 0, 1] <= top
[2, 1, 0] <= top
[2, 1] <= top
[2] <= top
[0, 1] <= top
[0] <= top

>>> num
3
```

Another Example: Binary Search

- To search a list of **n** items, first look at the item in location **n**/2
  - then search either the region from 0 to **n**/2-1
    or the region from **n**/2+1 to **n**-1
- Example: searching for 57 in a sorted list of 15 numbers



Recursive Version of Binary Search

- The full definition of recursive algorithm for binary search:

```
def rbsearch(a, k, lower = None, upper = None):
    lower = lower or -1      # assign -1 if lower = None
    upper = upper or len(a) # len(a) if upper = None
    mid = (lower + upper) // 2
    if mid == lower:
        return None
    if k == a[mid]:
        return mid
    if k < a[mid]:
        return rbsearch(a, k, lower, mid)
    if k > a[mid]:
        return rbsearch(a, k, mid, upper)
```

Non-recursive binary search with stack

```
def rbsearch_stack(a, k):
    lower = -1
    upper = len(a)
    s = Stack()
    s.push(lower)
    s.push(upper)

    while s.count() > 0:
        upper = s.pop()
        lower = s.pop()      Getting recursion parameters.
        mid = (lower + upper) // 2
        if mid == lower:
            return -1
        if k == a[mid]:
            return mid
        if k < a[mid]:
            s.push(lower)
            s.push(mid)      rbsearch(a, k, lower, mid)
        if k > a[mid]:
            s.push(mid)
            s.push(upper)    rbsearch(a, k, mid, upper)
```

In-Class Exercises

(a)

What is the output of calling **s.pop()** after the following operations?

```
s = Stack()
s.push(1)
s.push(3)
s.pop()
s.push(4)
s.pop()
s.push(2)
```

(b)

What is the state of the stack after the following operations?

```
s = Stack()
s.push(60)
s.peek()
s.push(34)
s.pop()
s.push(72)
s.push(44)
s.push(86)
s.pop()
s.pop()
s.push(59)
s.peek()

s.display()  ???
```