

Stacks, Queues, Linked-Lists

Instructor: Krishna Venkatasubramanian

CSC 212

Announcements

- Assignment 2 release Out.
 - Does not have visible test-cases
 - Scores will only be visible after the due date
- Assignment 2 Due date: Nov 26 11:59pm
- Next Quiz next Tuesday (Nov 12)

Dealing with data...

- How to use it ?
- How to **store** it ?
- How to **process** it ?
- How to gain “knowledge” from it ?
- How to keep it secret?

How should data be stored?

Depends on your requirement

Copyright 2005 by Randy Glasbergen.
www.glasbergen.com



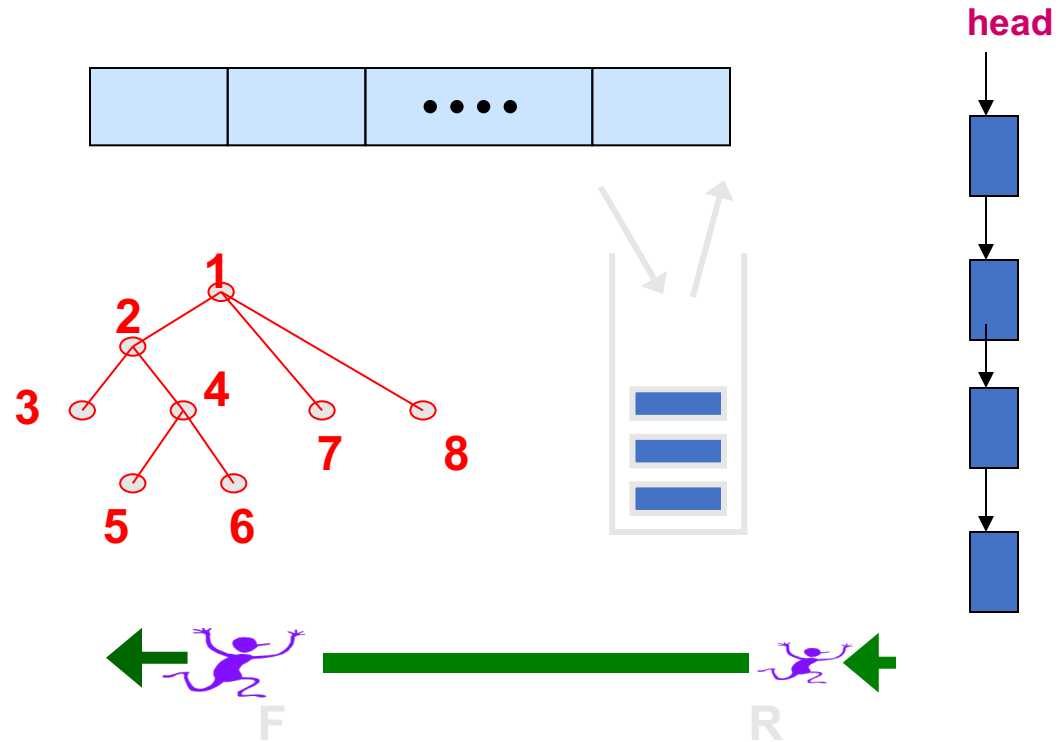
**“We back up our data on sticky notes because
sticky notes never crash.”**

Data is diverse ..
But we have some building blocks



Elementary Data “Structures”

- **Arrays**
- **Lists**
- **Stacks**
- **Queues**
- **Trees**



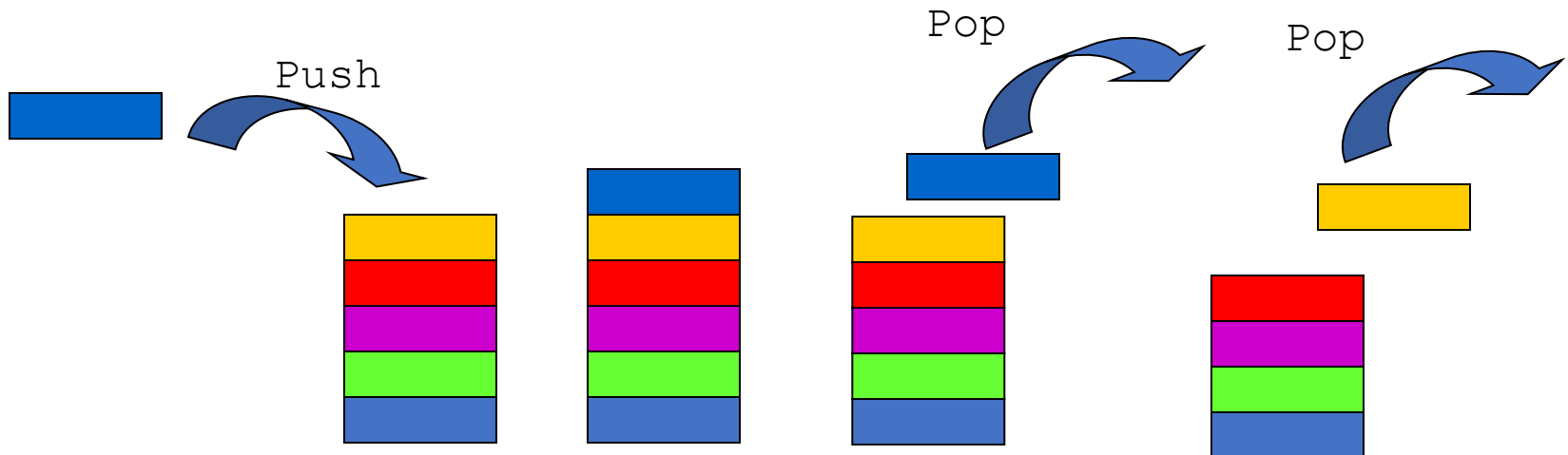
In some languages these are basic data types – in others they need to be implemented

Stacks

Stack

A list for which Insert and Delete are allowed only at one end of the list (the *top*)

- LIFO – Last in, First out



Stack Example: Postfix Calculation

- Postfix notation places operator after operands
 - Infix: $(3 + 2) * 10$
 - Postfix: $3\ 2\ +\ 10\ *$
- **Postfix Advantages:**
 - Do not need expressions to be parenthesized
 - Actually: It does not need any parentheses as long as each operator has a **fixed number of operands**.
 - Example:
 - An operator like “+ (addition)” needs two operands (numbers) that it can add.
 - $A+B$. here A and B are operands and + is the operator
 - Fewer operations need to be entered to perform typical calculations
 - Make fewer errors

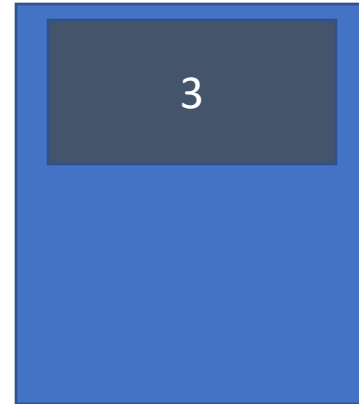
Stack Example: Postfix Calculation

- **Postfix notation** places operator after operands
 - Infix: $(3 + 2) * 10$
 - Postfix: $3\ 2\ +\ 10\ *$
- **A Stack can be used to calculate Postfix notations**
 - Push symbols as they appear
 - Whenever we read an operator, **pop two operands**
 - Evaluate operation, push result

Stack Example

A Stack Data Structure

3 2 + 10 *



Push element

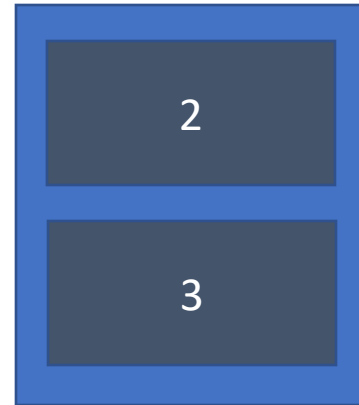
Stack Example

3 2 + 10 *



Push element

A Stack Data Structure

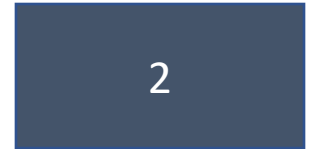
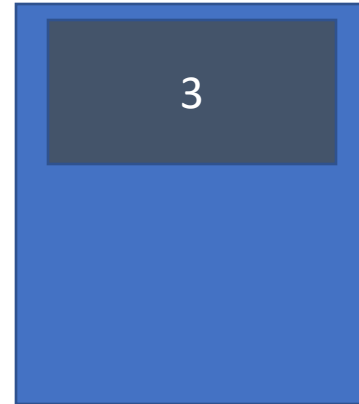


Stack Example

3 2 + 10 *



A Stack Data Structure



Pop two element

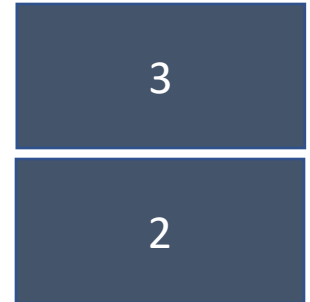
Stack Example

3 2 + 10 *



Pop two element


A Stack Data Structure



Stack Example

A Stack Data Structure

3 2 + 10 *



Perform operation and push result back

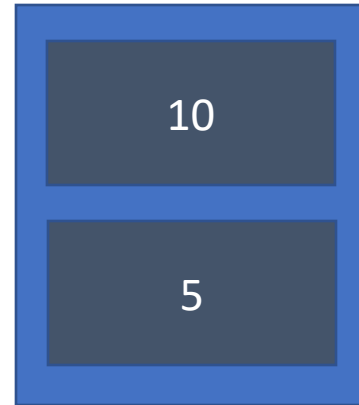
Stack Example

3 2 + 10 *



Push element

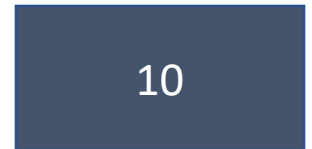
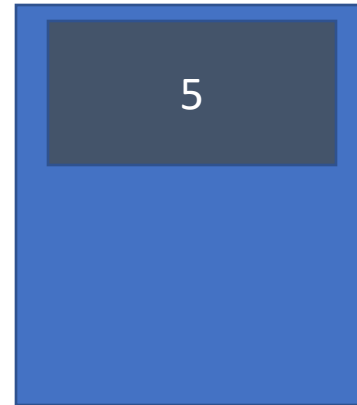
A Stack Data Structure



Stack Example

A Stack Data Structure

3 2 + 10 *



Pop two element

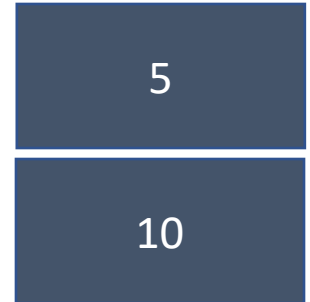
Stack Example

3 2 + 10 *



Pop two element

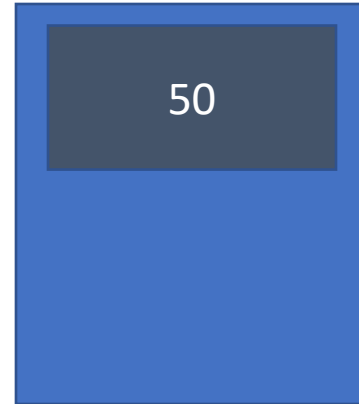
A Stack Data Structure



Stack Example

A Stack Data Structure

3 2 + 10 *



Perform operation and push result back

Exercise

- At home draw out the stack to solve the following Postfix expression

15 7 1 1 + - ÷ 3 × 2 1 1 + + -

- The answer you should get is 5

What else is this good for ?

- Page-visited history in a Web browser
- Undo sequence in a text editor
- Saving local variables when one function calls another, and this one calls another
- Matching parenthesis in text and code

Abstract data types and Data Structures

Data Structure

a way to store and organize data to facilitate access and modifications.

Ex. array, linked list, later in the course: hash table, heap, ...

Abstract Data Type (ADT)

a set of data values and associated operations that are precisely specified independent of any particular implementation.

Ex. stack, queue, ... later in the course: dictionary,...

- ADT describe the functionality of data structures
- Data structures are **implemented as** ADT
 - how is the data stored?
 - which algorithms implement the operations?

Abstract Data Types (ADTs)

- An abstract data type (ADT) is an abstraction of a data structure
- An ADT specifies:
 - Data stored
 - Operations on the data
 - Error conditions associated with operations

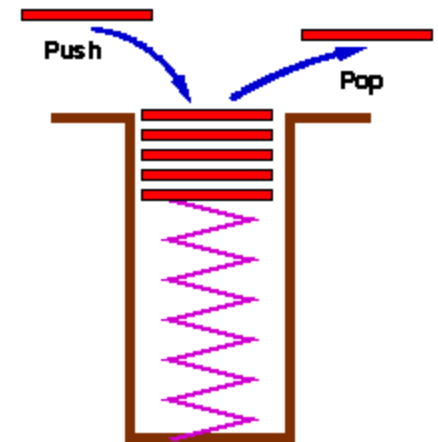
Stack ADT

Objects:

A finite sequence of nodes

Operations:

- create
- push: Insert element at top
- top: Return top element
- pop: Remove and return top element
- isEmpty: test if the stack is empty



Exceptions

- Attempting the execution of an operation of ADT may sometimes cause an error condition, called an exception
- Exceptions are said to be “thrown” by an operation that cannot be executed
- In the **Stack** ADT, operations **pop** and **top** cannot be performed if the stack is empty
- Attempting the execution of **pop** or **top** on an empty stack throws an **EmptyStackException**

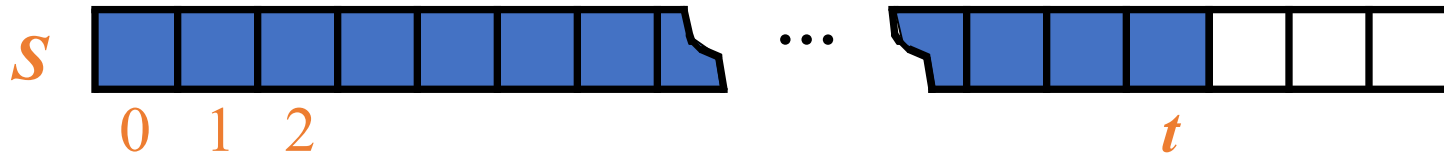
Exercise: Stacks

- Describe the output of the following series of stack operations
 - Push(8)
 - Push(3)
 - Pop()
 - Push(2)
 - Push(5)
 - Pop()
 - Pop()
 - Push(9)
 - Push(1)

Stack Visualization

<https://www.cs.usfca.edu/~galles/visualization/StackArray.html>

Array-Based Stack



- we add elements from left to right
- a variable keeps track of the index of the top element

```
class CapacitatedStack:
    def __init__(self):
        self.capacity = 20
        self.items = self.capacity * [None]
        self.count = 0
```

- capacity of stack is capped to size of array

Array-Based Stack

```
class CapacitatedStack:
    def __init__(self):
        self.capacity = 20
        self.items = self.capacity * [None]
        self.count = 0
```

- How can we implement the operations **size**, **isEmpty**, **push** and **pop**?

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

```
def isEmpty(self):
    return self.count == 0
```

```
def push(self, item):
    if self.count < self.capacity:
        self.items[self.count] = item
        self.count += 1
    else: raise Exception("CapacitatedStack overflow.")
```

```
def pop(self):
    if self.count > 0:
        self.count -= 1
        return self.items[self.count]
    else:
        raise Exception("Cannot pop from empty stack")
```

What are running times of these operations?

Growable Array-based Stack

- Fixed-capacity stack: fast but not very useful
- How can we make an array-based stack that has unlimited capacity?
 - **Incremental strategy**: increase the size of the array by a constant c when capacity is reached
 - **Doubling strategy**: double the size of the array when capacity is reached
- **Problem: arrays cannot be resized. You can only copy over elements to a new array**

Growable Array-based Stack

```
def push(self, item):
    if self.count >= self.capacity:
        # doubling, incremental would be += ...
        self.capacity *= 2
        copy_items = self.capacity*[None]
        for i in range(self.count): copy_items[i] = self.items[i]
        self.items = copy_items

    self.items[self.count] = item
    self.count += 1
```

What's the runtime of push?

- when the stack doesn't expand?

$O(1)$

- when it does expand?

Incremental: **$O(n)$**

Doubling: **$O(n)$**

WHY? : need to copy the current elements in the stack (array) to the new one

Stacks/Growable Arrays in Python

- The list data type in Python is based on a growable array with doubling strategy --- can be used to build Stack ADT

<code>s.append(x)</code>	PUSH	Can be used to implement appends <code>x</code> to the end of the sequence, here, end of the list is the top of the stack
<code>s.pop()</code>	POP	Retrieves and removes the item from the end of list
<code>s == []</code>	IS_EMPTY	Checks if stack is empty
<code>s[-1]</code>	TOP	Gets the item from the end of the list

- The Stack ADT operations :
 - push/ isEmpty/ top / pop can be performed in $\rightarrow O(1)$

Complexity of implementing Python data structures

https://www.ics.uci.edu/~brgallar/week8_2.html

Queues



Queues

Queue

Stores a set S of elements with insertions and deletions follow a **FIFO** (first-in, first-out) scheme

Operations for a Queue ADT

enqueue(S, x): inserts element x into S

dequeue(S): removes and returns the element first inserted into S

size(S): returns the number of elements in S

isEmpty(S): indicates whether S is the empty set

Growable Array-based Queue

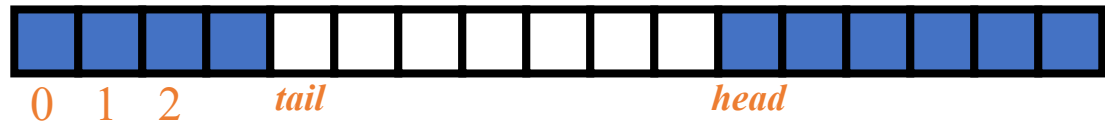
- We can also implement a queue using an growing array, but with a slight complication
- Unlike a stack, we need to **keep track of the head and the tail** of the queue



- What happens if the tail reaches the end of the array, but there's still room at the front? Is the queue full?

Growable Array-based Queue

- **Wrap the queue!**



- **Expand the array when queue is completely full**

- When copying, “unwind” the queue so the head starts back at 0

enqueue(x):

```
if size == capacity:  
    double array and copy contents  
    reset head and tail pointers  
data[tail] = x  
tail = (tail + 1) % capacity  
size++
```

dequeue():

```
if size == 0:  
    error("queue empty")  
element = data[head]  
head = (head + 1) % capacity  
size--  
return element
```



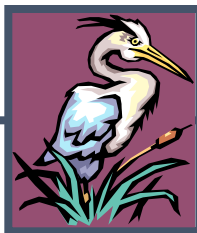
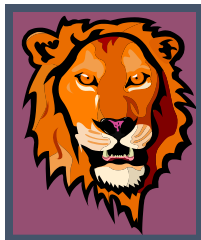
Queues in Python

- Do Python lists provide an efficient implementation of Queues if used “directly”?
- No:
 - enqueue(x): s.append(x) in O(1) time, but
 - dequeue(): s.pop(0) in O(n) time **[Not the same as s.pop() used in a stack. Notice the index in the call to pop]**
- Deques (double-ended Queues) are provided as collections.deque

```
from collections import deque
S = deque([2, 3, 5])
S.append(7)
S.popleft()
S
deque([3, 5, 7])
```

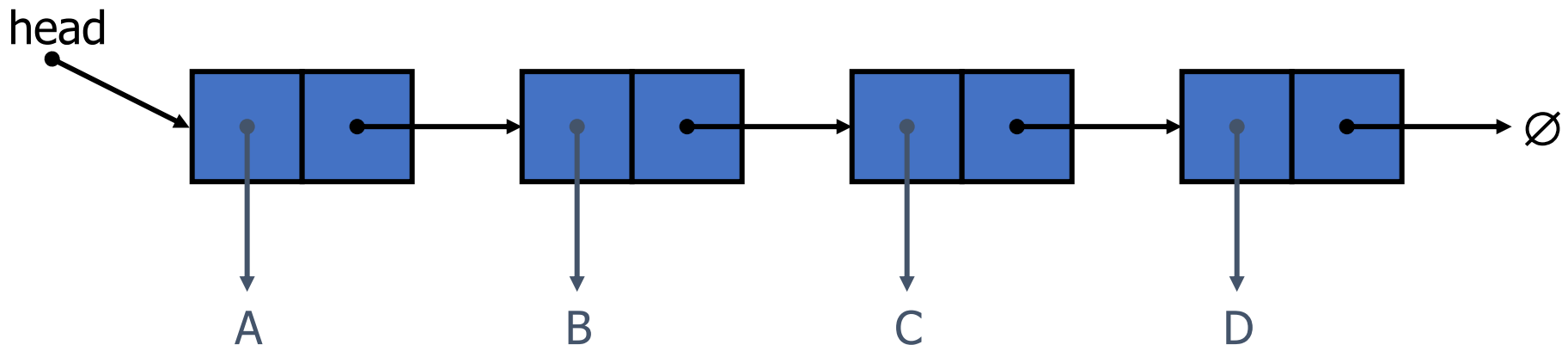
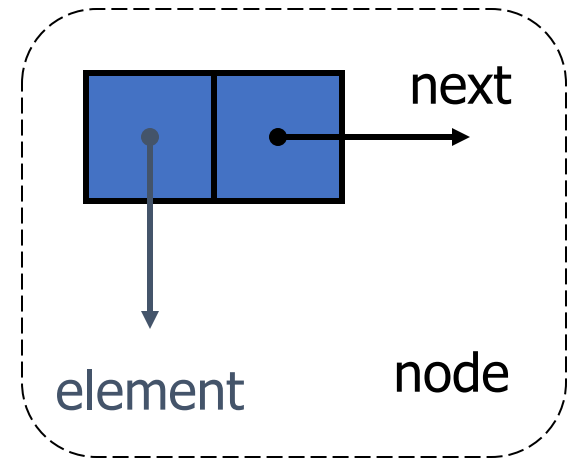
- deques are not implemented using arrays but doubly-linked lists

Linked Lists



Singly Linked List

- singly linked list: data structure consisting of
 - a sequence of nodes,
 - starting from a head pointer
- each node stores
 - element
 - **link to the next node** (use pointers with C/C++)
- Nodes are not adjacent in memory like an array
- **Elements** of nodes can be combination of multiple data-types unlike arrays!



Implementing a Singly Linked List

```
class SNode:
    def __init__(self, elem=None, next=None):
        self.elem = elem
        self.next = next

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

```
node3 = SNode("Toronto")
node2 = SNode("Seattle", node3)
node1 = SNode("Rome", node2)

list = SList()
list.head = node1

currentNode = list.head
while (currentNode):
    print(currentNode.elem)
    currentNode = currentNode.next
```

Rome
Seattle
Toronto

How do we insert/delete efficiently in a Singly Linked List?

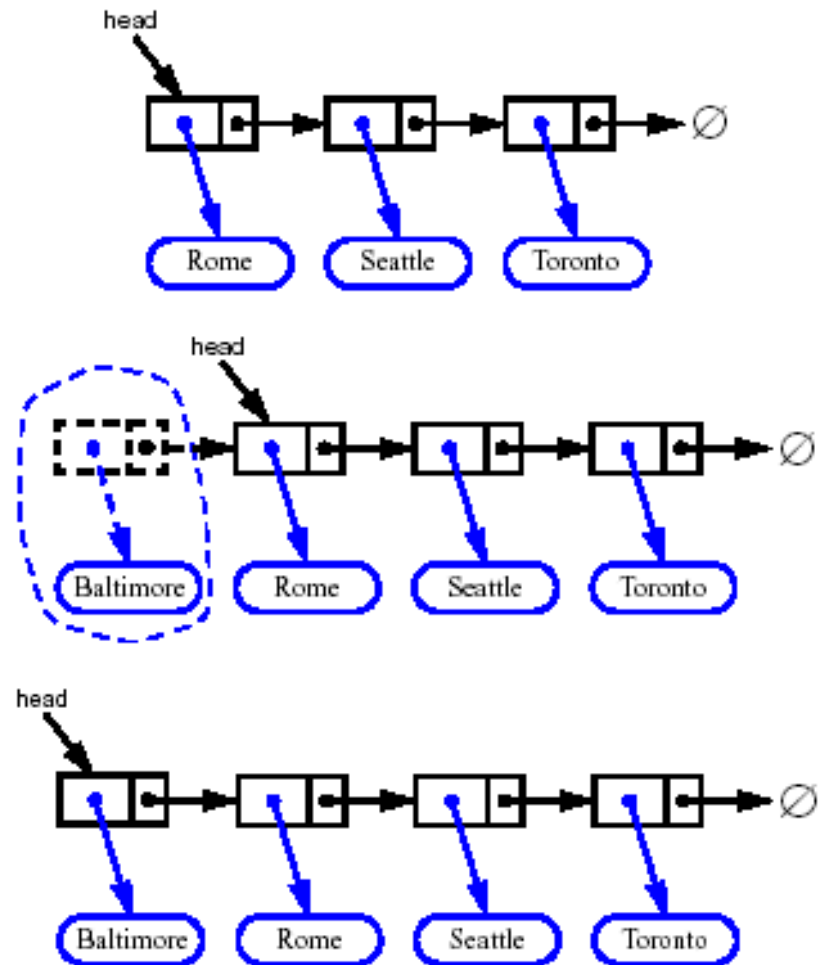
Inserting at the Head

1. Allocate a new node
2. Insert new element
3. Have new node point to old head
4. Update head to point to new node

```
newNode = SNode("Baltimore")
newNode.next = list.head
list.head = newNode

printList(list)
```

```
Baltimore
Rome
Seattle
Toronto
```

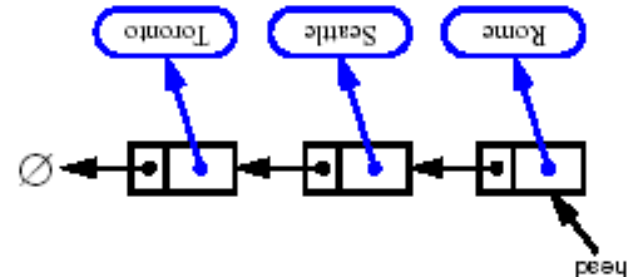
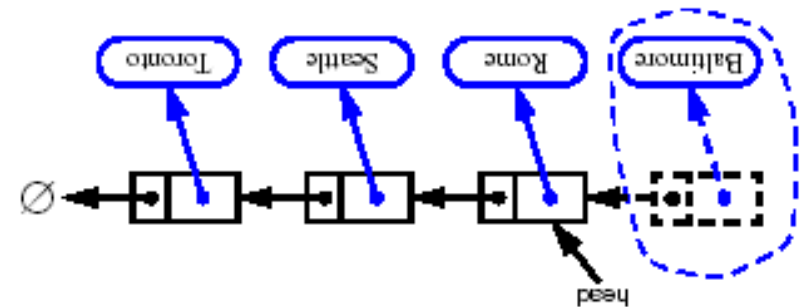
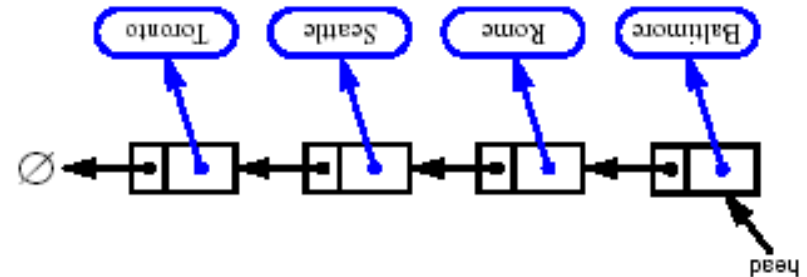


Removing at the Head

1. Update head to point to next node

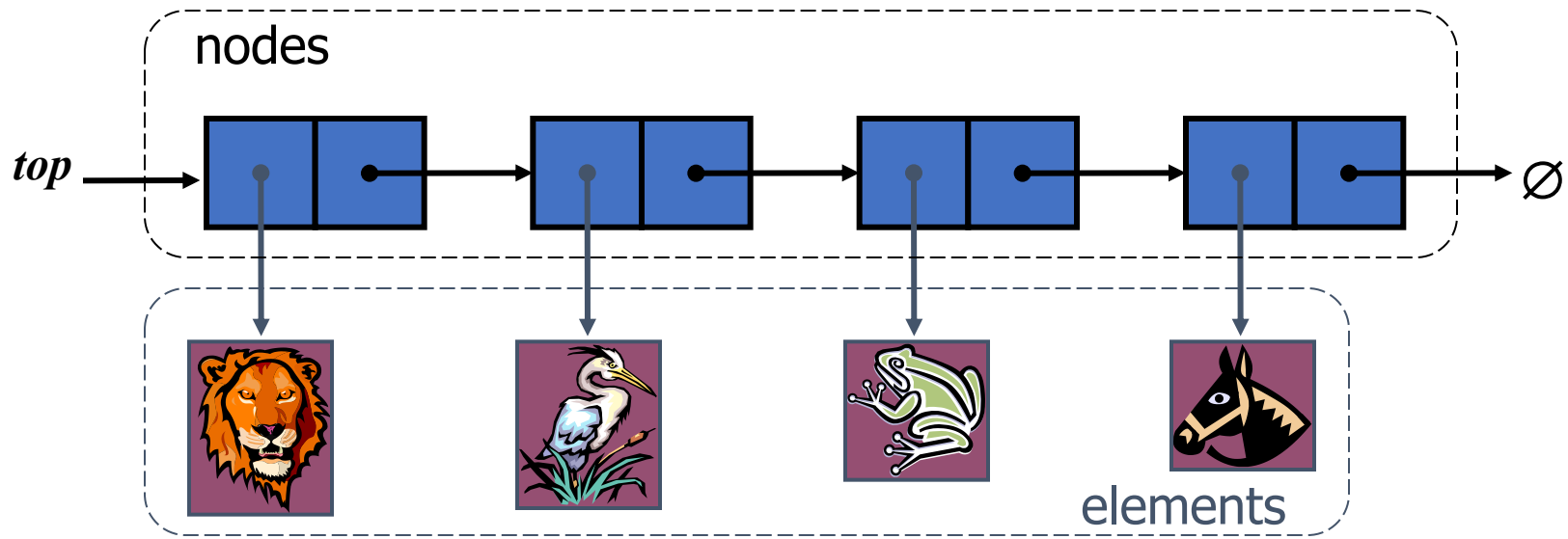
```
list.head = list.head.next  
printList(list)
```

Rome
Seattle
Toronto



Stack as Singly Linked List

- top element at head (i.e., the head is called top, when we are dealing with stack)



- The space used is $O(n)$ and each operation of the Stack ADT takes $O(1)$ time

Stack as a Singly Linked List

```
class Stack:
    def __init__(self):
        self.list = SList()
        self.count = 0

    def isEmpty(self):
        return self.count == 0

    def push(self, item):
        newNode = SNode(item, self.list.head)
        self.list.head = newNode
        self.count += 1

    def pop(self):
        if self.isEmpty():
            raise Exception('stack is empty.')
        else:
            item = self.list.head.elem
            self.list.head = self.list.head.next
            self.count -= 1
            return item

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

Stack: Defined in terms of the Singly Linked List

Push: Add to the head of the list

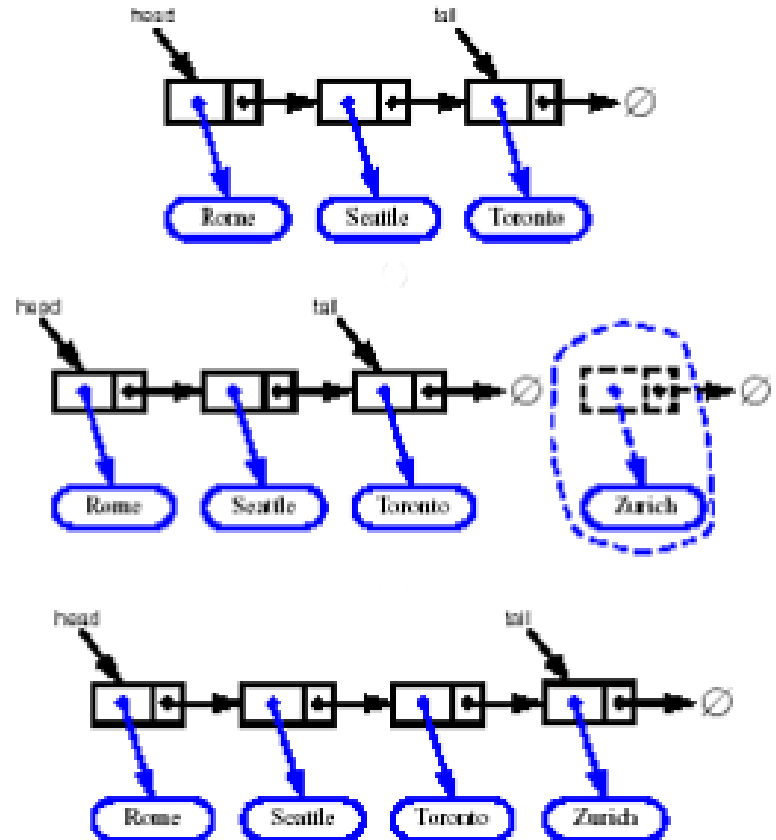
Pop: Remove to the head of the list
Move the head

Inserting at the Tail (at the End)

1. Allocate a new node
2. Insert new element
3. Have new node point to null
4. Have old last node point to new node
5. Update tail to point to new node

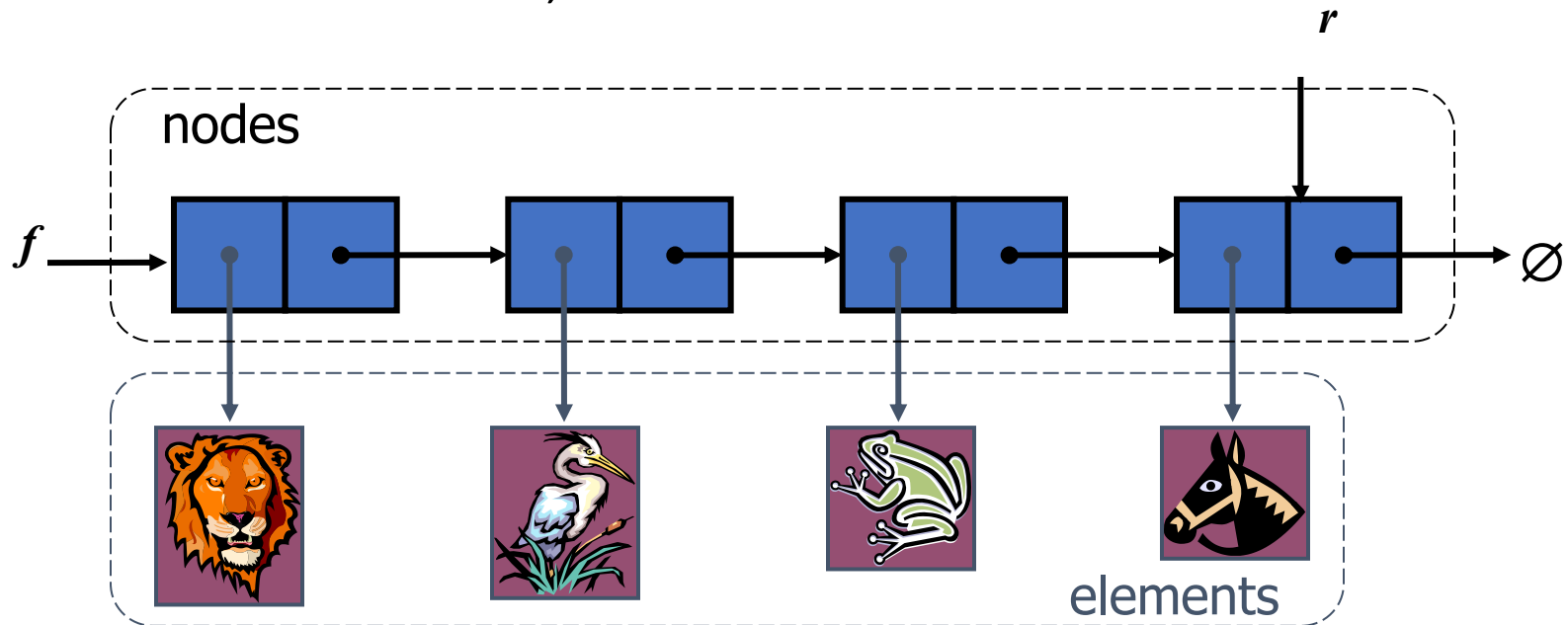
- **requires pointer to tail:
list.tail**

- Complexity of appending to tail = $O(1)$



Queue as Singly Linked List

- front element at head, rear element at tail



- The space used is $O(n)$ and each operation of the Queue ADT takes $O(1)$ time

Queue as Singly Linked List

```
class Queue:
    def __init__(self):
        self.head = None
        self.tail = None
        self.count = 0

    def isEmpty(self):
        return self.count == 0

    def enqueue(self, elem):
        newNode = SNode(elem)
        if self.isEmpty():
            self.head = newNode
            self.tail = newNode
        else:
            self.tail.next = newNode
            self.tail = newNode
        self.count += 1
```

```
    def dequeue(self):
        if self.isEmpty():
            raise Exception('Queue is empty.')
        else:
            elem = self.head.elem
            if self.head.next:
                self.head = self.head.next
            else:
                self.head = None
                self.tail = None
            self.count -= 1
            return elem

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

```
Q = Queue()
Q.isEmpty()
```

True

```
Q.enqueue('lion')
Q.enqueue('bird')
Q.enqueue('frog')
Q.enqueue('horse')
Q.dequeue()
```

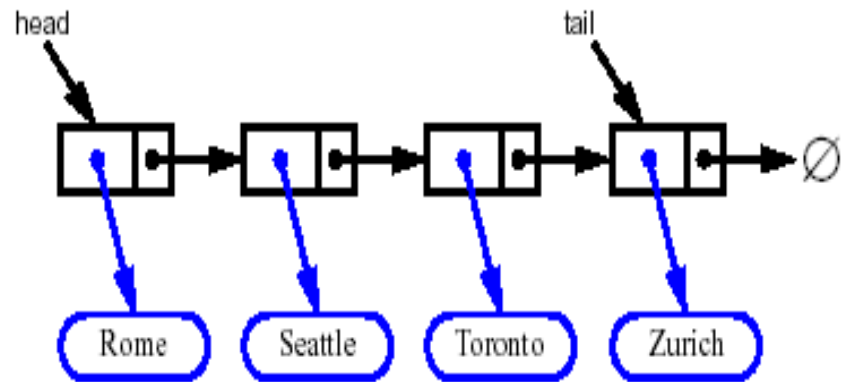
'lion'

**How do we implement a Deque
(pop/push at both ends)?**

Removing at the Tail ?!

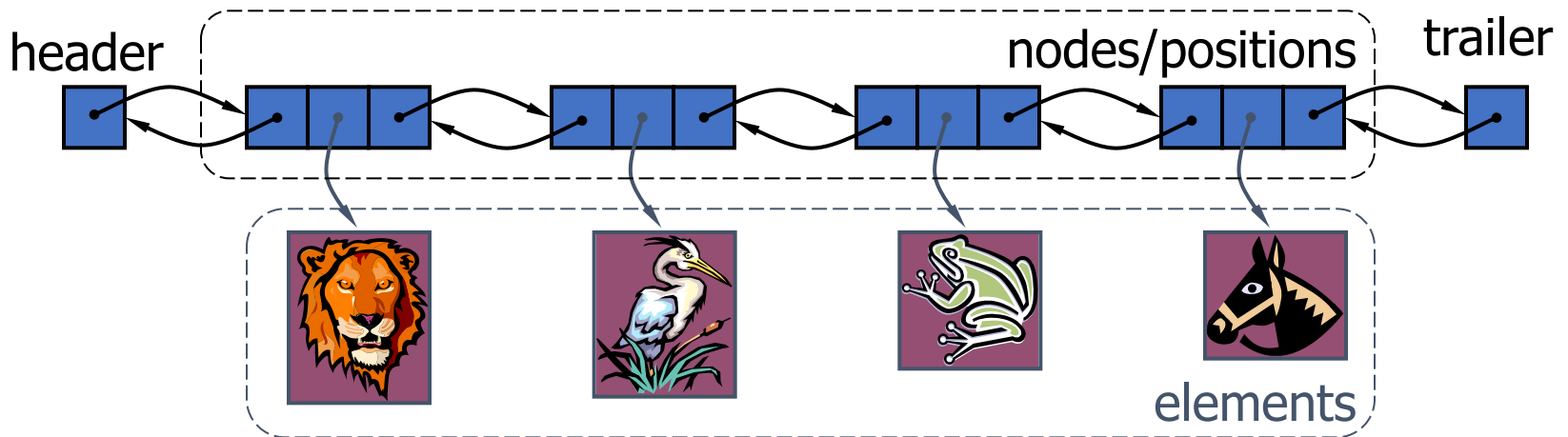
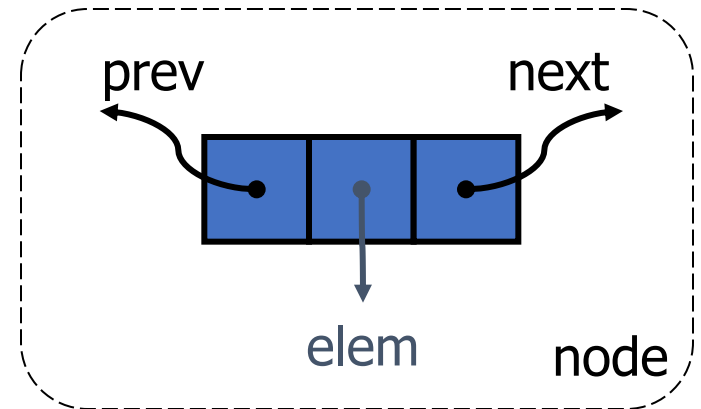
- **no constant-time way to update the tail to point to the previous node. WHY?**

- removing at the tail of a singly linked list is not efficient!
 - Once has to traverse the whole length of the list
 - Complexity $O(n)$



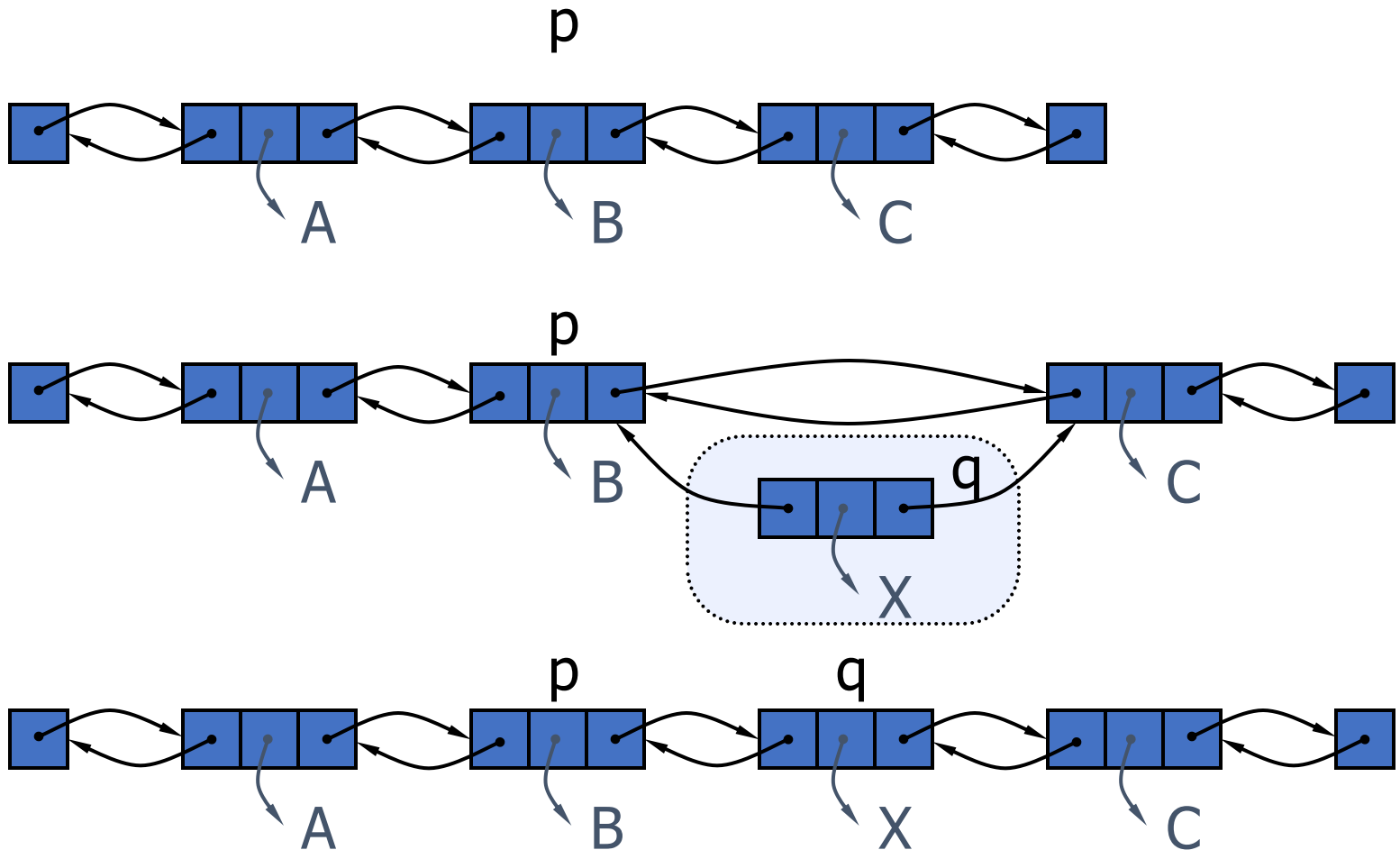
Doubly Linked List

- each node stores
 - element
 - link to next node
 - **link to previous node**
- special trailer and header nodes
- **makes adding and remove at both ends of the list $O(1)$ complexity**
 - Perfect for dequeue in Python



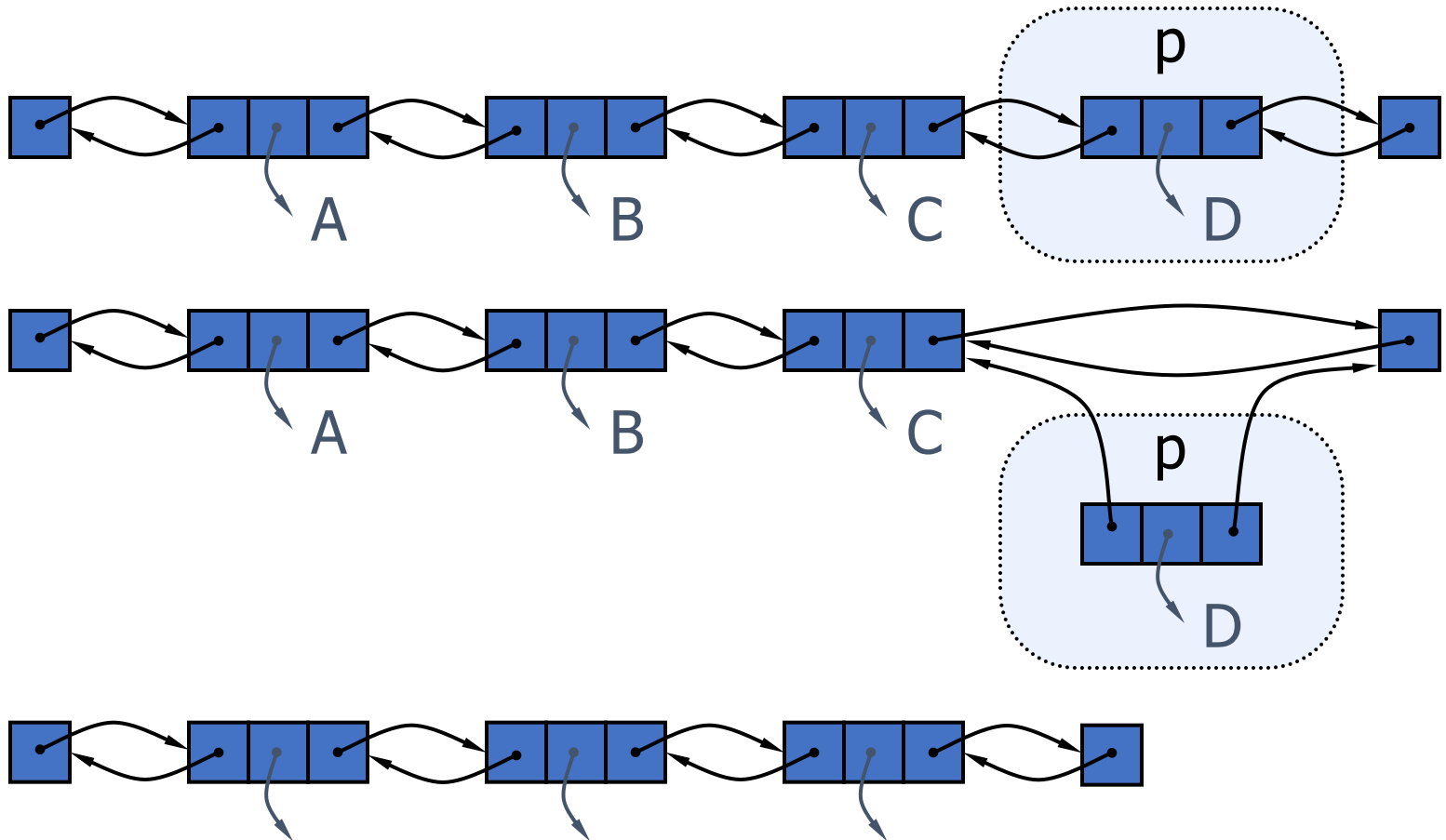
Insertion

- Insert a new node, q , between p and its successor.



Deletion

- Remove a node, p , from a doubly-linked list.



Summary

- ADTs we have seen so far
 - Stacks, Queues, (and Lists and Sets in Python)
- data structures for Stacks and Queues:
 - array, but fixed capacity
 - linked lists
- Singly Linked Lists vs Doubly Linked Lists



That's all Folks!
Any Question?