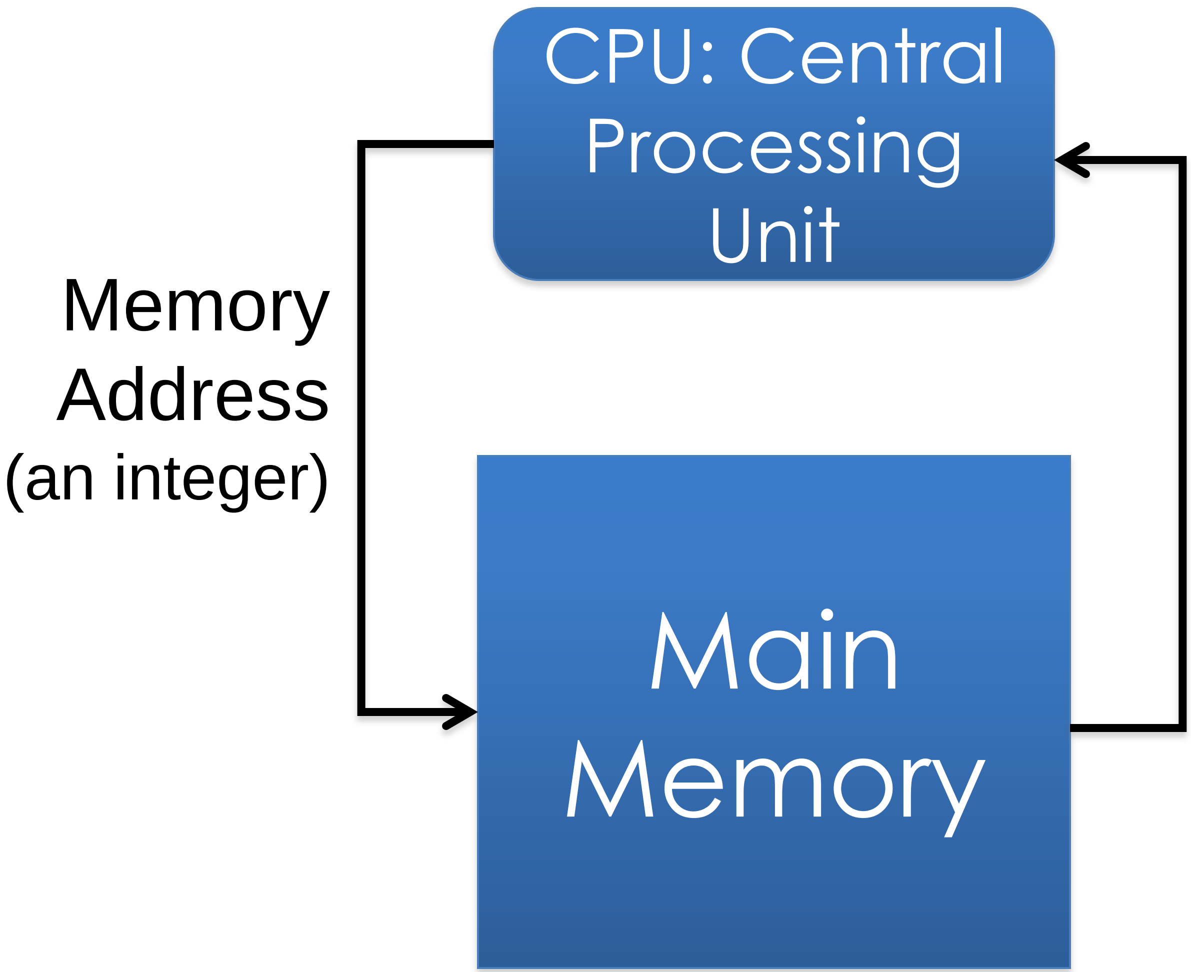
Organizing Data:
Arrays, Linked Lists

Computer Memory



Recall Lists

- *Ordered* ***collection*** of data
- Our mental model is based on indexed data slots

0	1	2	3	4	5	6	7	8
A	B	C	D	E	F	G	H	I

- But how are lists actually stored in computer's memory?

Organizing Data in Memory

- We are going to see in a few weeks how data types such as integers, strings are represented in computer memory as sequence of bits (0s, 1s).
- We will work at a higher-level of abstraction and talk about how **collections of data** are organized in memory.
  - For example, how are Python lists organized in memory? How could we organize our data to capture hierarchical relationships between data?

Data Structure

- The organization of data is a very important issue for computation.
- A **data structure** is a way of storing data in a computer so that it can be used efficiently.
  - Choosing the right data structure will allow us to develop certain algorithms for that data that are more efficient.

Today's Lecture

- Two basic structures for ordered sequences:
  - Arrays and
  - Linked lists

Arrays in Memory

- An **array** is a very simple data structure for holding a **sequence of data**. They have a direct correspondence with memory system in most computers.
- Typically, array elements are **stored in adjacent memory cells**. The subscript (or index) is used to calculate an offset to find the desired element.

Address	Content
100:	50
104:	42
108:	85
112:	71
116:	99

Example: data = [50, 42, 85, 71, 99]

Assume we have a byte-addressable computer,
* **integers** are stored using **4 bytes** (32 bits)
* the **first** element is stored at address 100
(Nothing special about 100, just an example).

The array could start at any address.

Arrays in Memory

- Example: data = [50, 42, 85, 71, 99]
  Assume we have a byte-addressable computer, integers are stored using 4 bytes (32 bits) and our array starts at address 100.

- If we want data[3], the computer takes the address of the start of the array (100 in our example) and adds **the index * the size** of an array element (4 bytes in our example) to find the element we want.

Location of data[3] is 100 + 3*4 = 112

- **Do you see why it makes sense for the first index of an array to be 0?**

	Content
100:	50
104:	42
108:	85
112:	71
116:	99

Arrays: Pros and Cons

- Pros:
  - **Access** to an array element **is fast** since we can compute its location quickly (constant time).
- Cons:
  - If we want to **insert or delete** an element, we have to **shift** subsequent elements **which slows** our computation down.
  - We need a **large enough block of memory** to hold our array.

Arrays in Python

- Array module
- Arrays are sequence types and behave very much like lists, except that the **type of objects** stored in them is constrained.
- We only use Python lists in 15110.
  Python lists are akin to structures called **dynamic arrays.**

Linked Lists

- Another data structure that stores a sequence of data values is the **linked list**.
- Data values in a linked list do **not have to be** stored in **adjacent** memory cells.
- To accommodate this feature, each data value has an additional "***pointer***" that indicates where the *next data value* is in computer memory.
- In order to use the linked list, we only need to know where the *first data value* is stored.

Linked List Example

- Linked list to store the sequence: data = [50, 42, 85, 71, 99]

Assume each integer and each pointer requires 4 bytes.

Starting Location of List (head)
124

	data	next
100:	42	148
108:	99	0 (null)
116:		
124:	50	100
132:	71	108
140:		
148:	85	132
156:		

Linked List Example

To insert a new element,
we only need to change a few pointers.

Example:
Insert 20 between
42 and 85

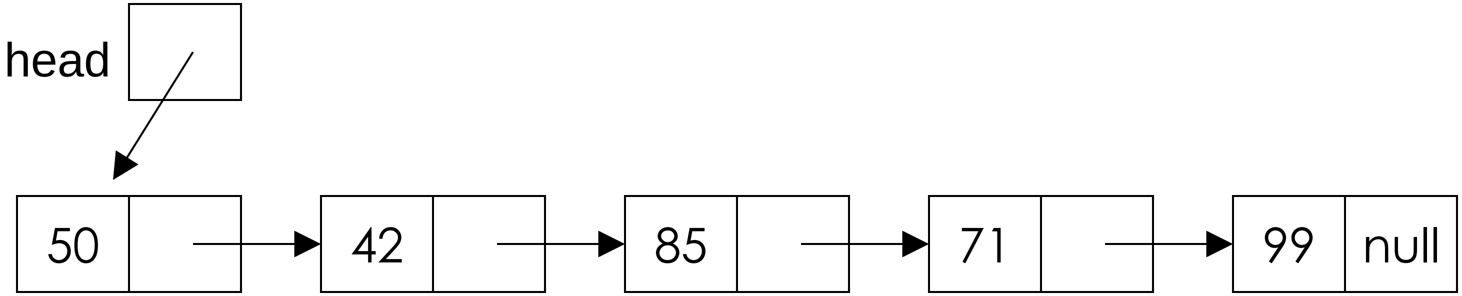
Starting Location of List (head)
124

Assume each integer and pointer requires 4 bytes.

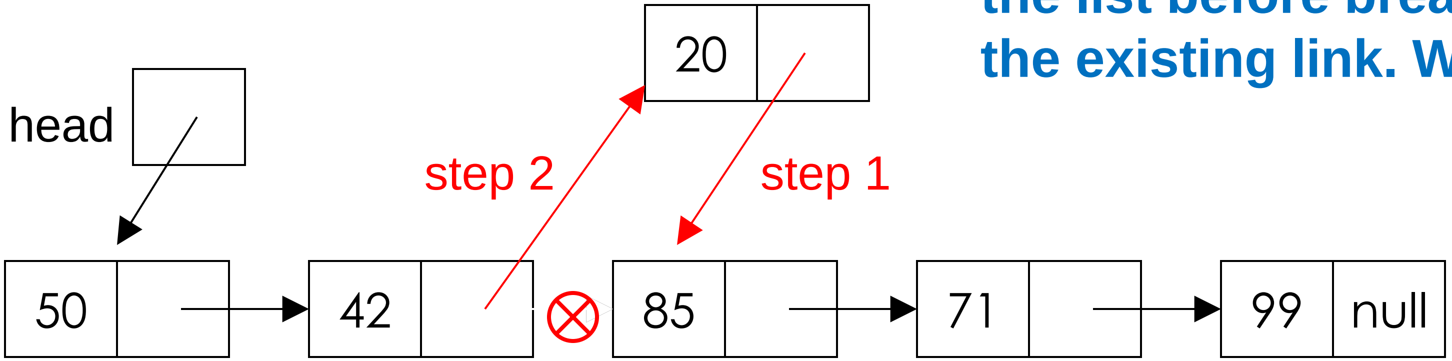
	data	next
100:	42	156
108:	99	0 (null)
116:		
124:	50	100
132:	71	108
140:		
148:	85	132
156:	20	148

Drawing Linked Lists Abstractly

[50, 42, 85, 71, 99]



Inserting 20 after 42:



We link the new node to the list before breaking the existing link. Why?

Linked Lists: Pros and Cons

- Pros:
  - **Inserting** and **deleting** data does not require us to move/shift subsequent data elements.
- Cons:
  - If we want to **access** a specific element, we **need to traverse** the list from the head of the list to find it, which can take longer than an array access.
  - Linked lists require **more memory**. **(Why?)**

Two-dimensional arrays

- Some data can be organized efficiently in a **table** (also called a **matrix** or **2-dimensional array**)
- Each cell is denoted with two subscripts, a **row** and **column** indicator

B[2][3] = 50

B	0	1	2	3	4
0	3	18	43	49	65
1	14	30	32	53	75
2	9	28	38	**50**	73
3	10	24	37	58	62
4	7	19	40	46	66

2D Lists in Python

```
data = [ [1, 2, 3, 4],
         [5, 6, 7, 8],
         [9, 10, 11, 12]
       ]
>>> data[0]
[1, 2, 3, 4]
>>> data[1][2]
7
>>> data[2][5]
index error
```

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

2D List Example in Python

- Find the sum of all elements in a 2D array

```
def sum_matrix(table):
    sum = 0
    for row in range(0,len(table)):
        for col in range(0,len(table[row])):
            sum = sum + table[row][col]
    return sum
```

number of rows in the table

number of columns in the given row of the table

In a rectangular matrix, this number will be fixed so we could use a fixed number for row such as len(table[0])

Tracing the Nested Loop

```
def sum_matrix(table):
    sum = 0
    for row in range(0,len(table)):
        for col in range(0,len(table[row])):
            sum = sum + table[row][col]
    return sum
```

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

len(table) = 3
len(table[row])= 4 for every row

row	col	sum
0	0	1
0	1	3
0	2	6
0	3	10
1	0	15
1	1	21
1	2	28
1	3	36
2	0	45
2	1	55
2	2	66
2	3	78

Stacks

- A **stack** is a data structure that works on the principle of **Last In First Out** (LIFO).
  - LIFO: The last item put on the stack is the first item that can be taken off.
- Common stack operations:
  - Push – put a new element on to the top of the stack
  - Pop – remove the top element from the top of the stack

- Applications: calculators, compilers, programming

RPN

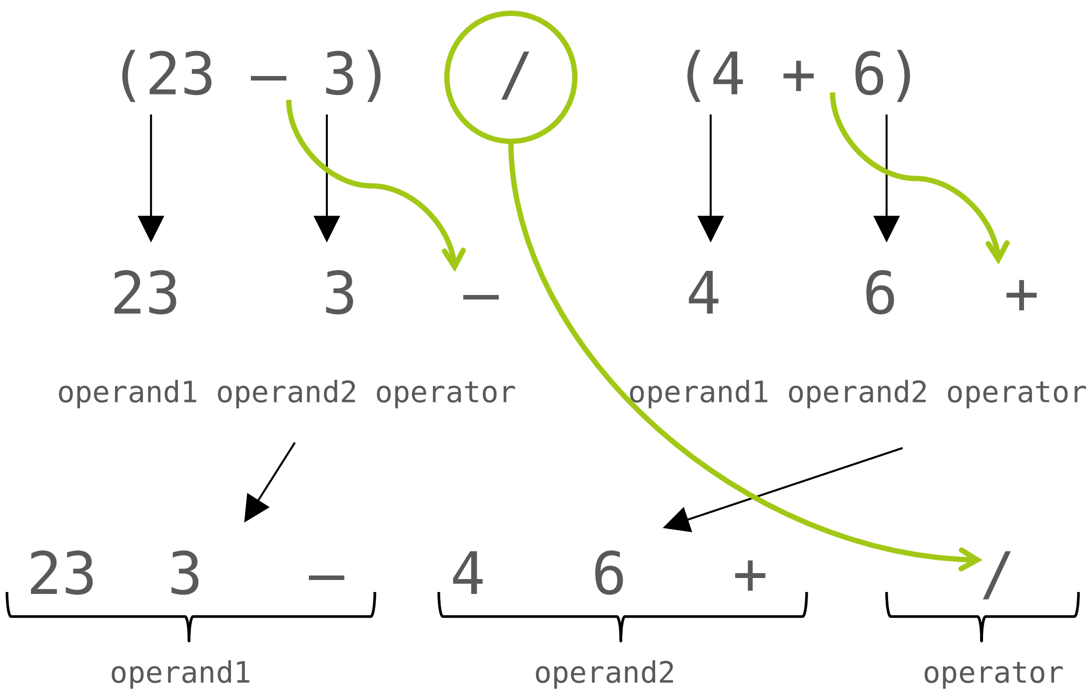
- Some modern calculators use Reverse Polish Notation (RPN)
  - Developed in 1920 by Jan Lukasiewicz
  - Computation of mathematical formulas can be done without using any parentheses

Example:

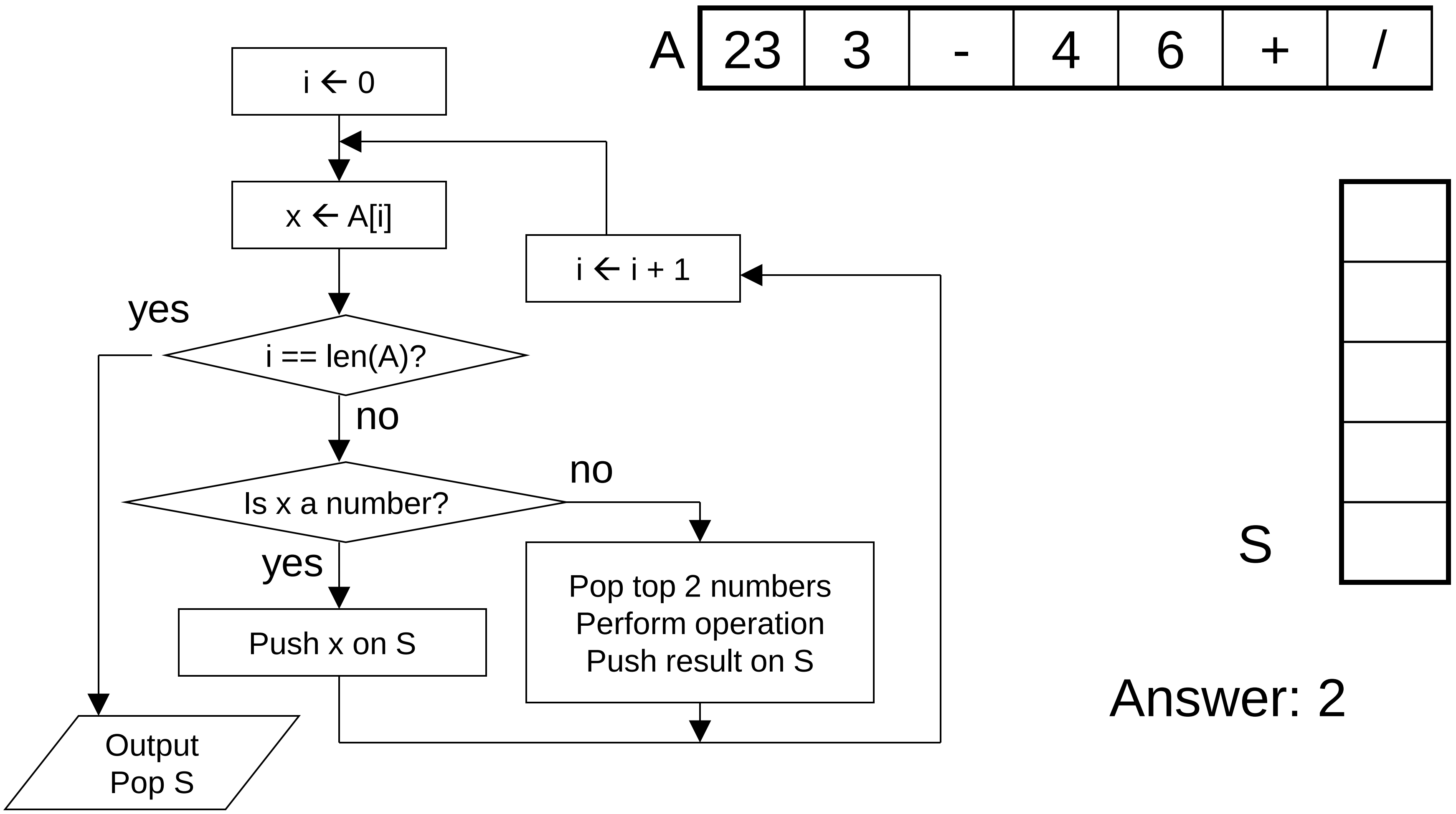
In RPN `(3 + 4) * 5`
becomes `3 4 + 5 *`

RPN Example

Converting a standard mathematical expression into RPN:



Evaluating RPN with a Stack



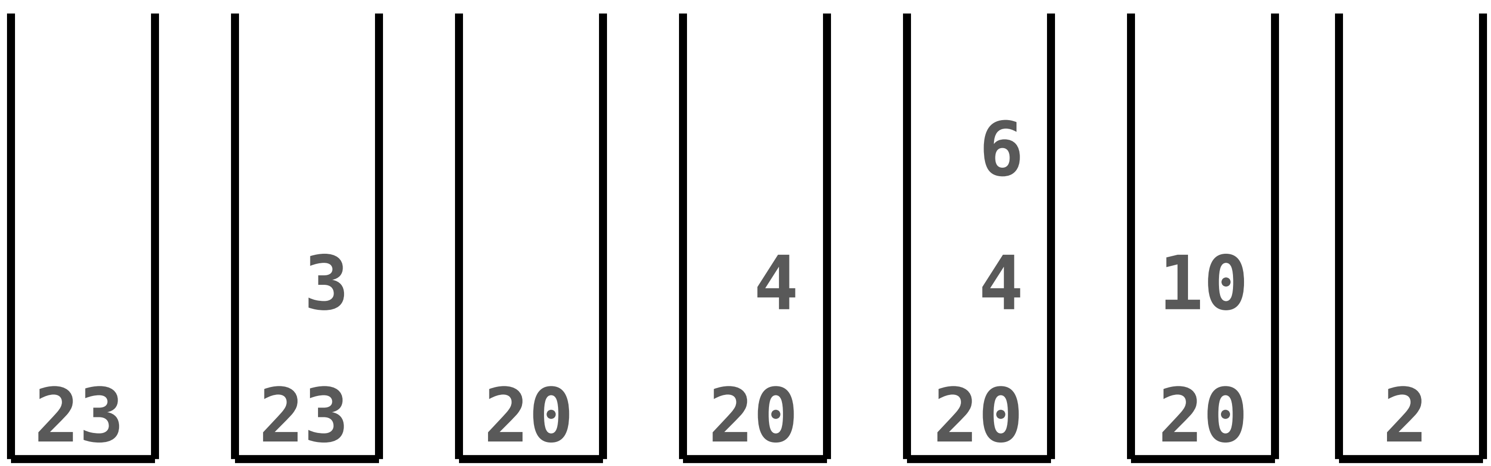
Answer: 2

Example Step by Step

RPN:

23 3 - 4 6 + /

Stack Trace:



Stacks in Python

You can treat lists as stacks in Python.

	stack	**x**
`stack = []`	[]	
`stack.append(1)`	[1]	
`stack.append(2)`	[1,2]	
`stack.append(3)`	[1,2,3]	
`x = stack.pop()`	[1,2] 3	
`x = stack.pop()`	[1]	2
`x = stack.pop()`	[]	1
`x = stack.pop()`	[]	ERROR

Queues

A **queue** is a data structure that works on the principle of **First In First Out** (FIFO).

- FIFO: The first item stored in the queue is the first item that can be taken out.

Common queue operations:

- Enqueue – put a new element in to the rear of the queue
- Dequeue – remove the first element from the front of the queue

Applications: printers, simulations, networks

Next Time

Hash Tables