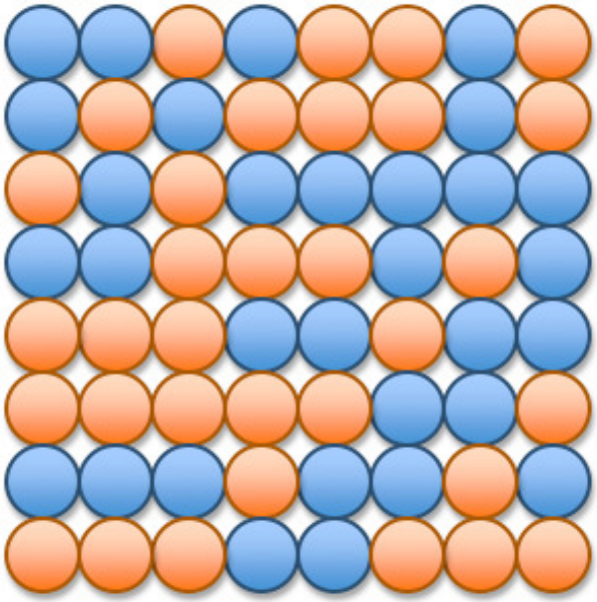


UNIT 6A
Organizing Data

Announcements

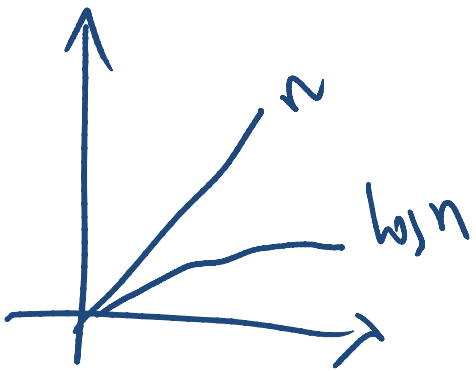
- We will be splitting the first lecture into two groups for the exam. My slides from last week has the exact information about who needs to go where for the exam.
  - **2:30 Exam:**
    - Sections A, B, C, D, E go to **Rashid (GHC 4401)**
    - **Sections F, G go to PH 125C.**
  - **3:30 Exam:**
    - Sections H, I, J, K, L, M, N all go to **Rashid (GHC 4401).**
  - Bring your CMU id!
- People who need extended time should confirm their arrangements with Dilsun
  - dilsun@cs.cmu.edu

Data Explosion

- The data on Internet doubles every 6 months
- Challenge is store the data so they can be searched easily
- What are some algorithms for searching data?

— linear search $O(n)$

— binary search $O(\log n)$



Data Processing Challenges

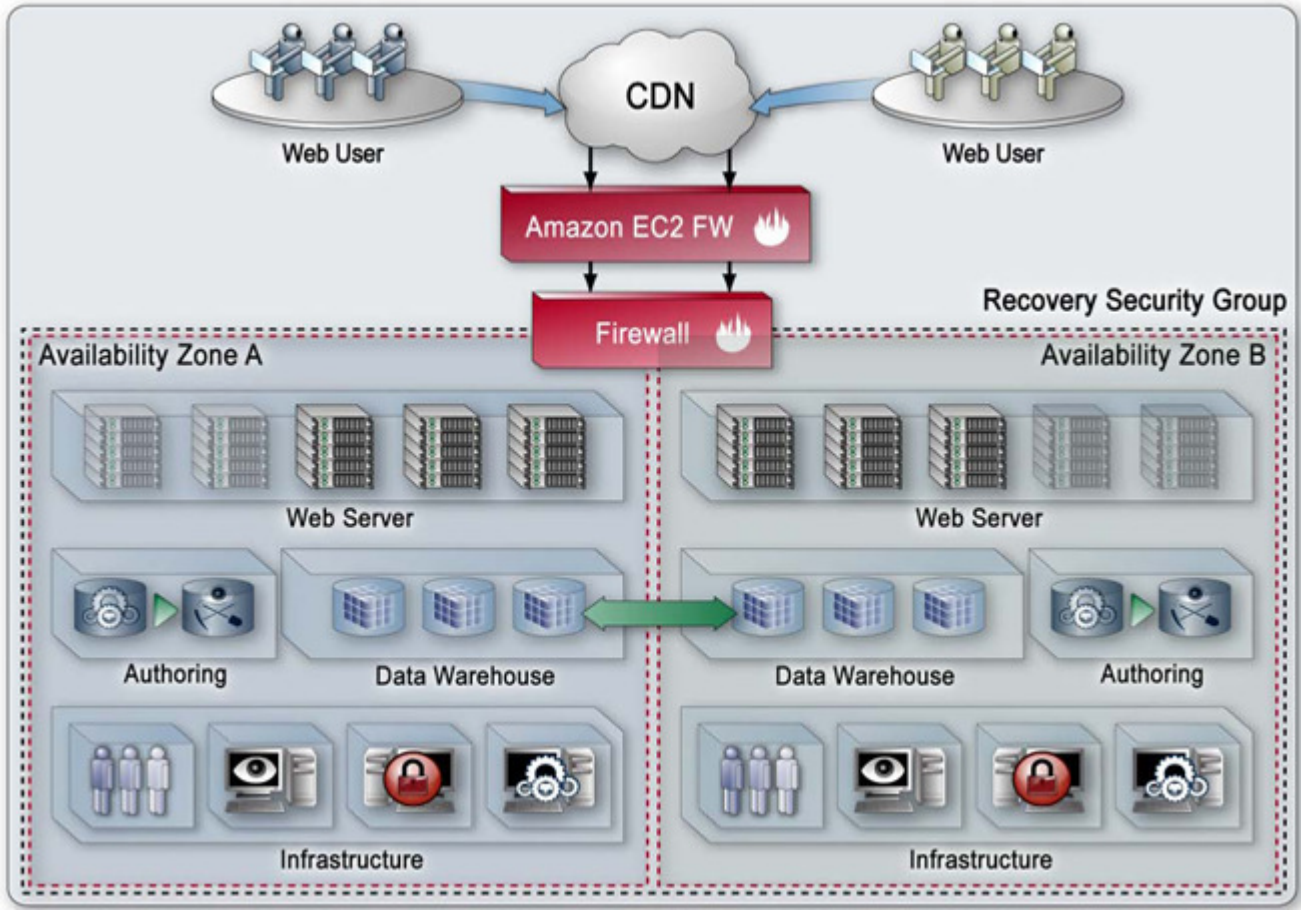
- Sort million records in a fraction of a second
- Build a relationship graph from a known set of relationship pairs
- Find the shortest distance from A to B
- Find all people who are in proximity to me
- What are some others?

How are data stored to meet these challenges

Local Storage Devices

Images from many public sources

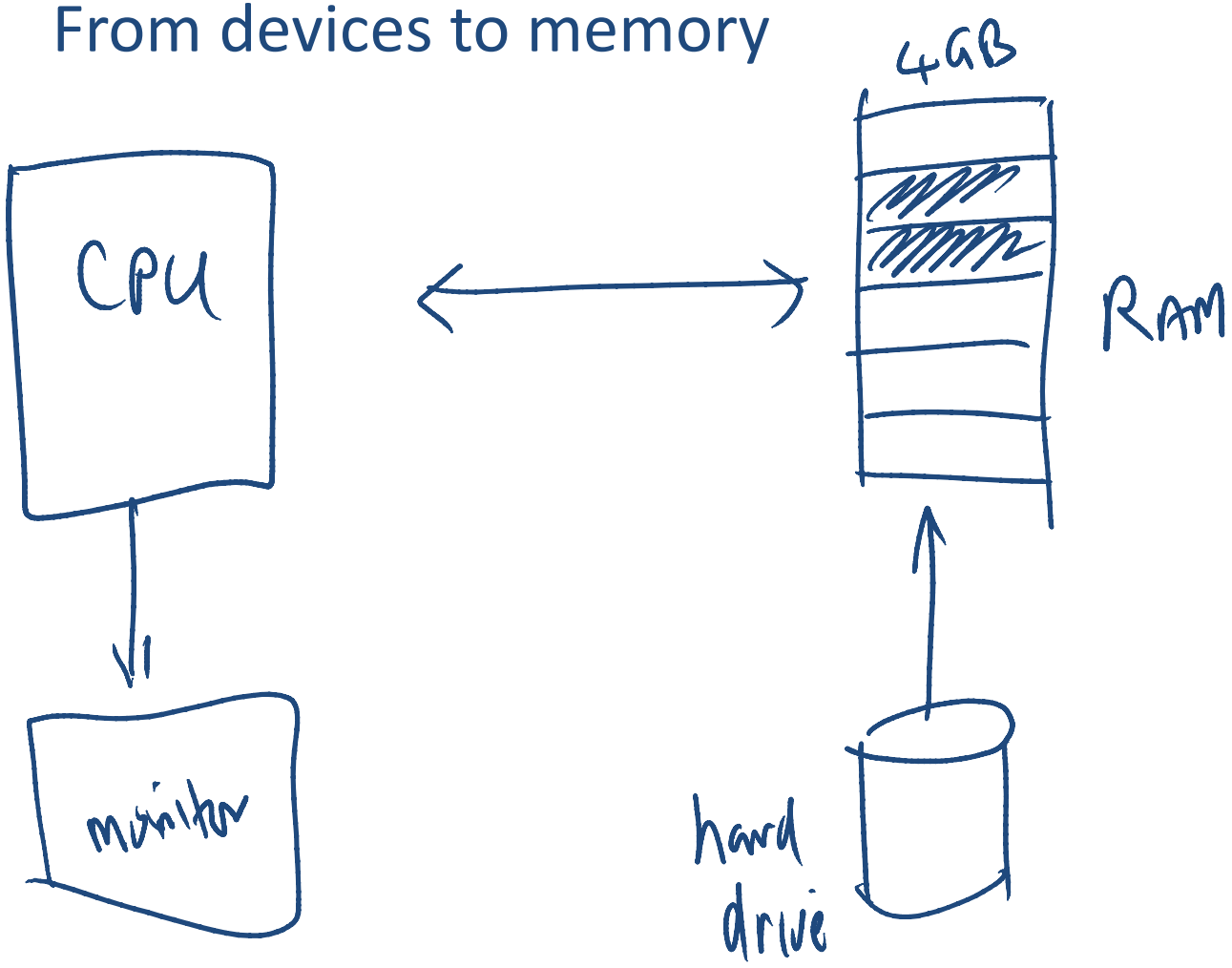
A Cloud Architecture



Source: Amazon.com

How is data processed?

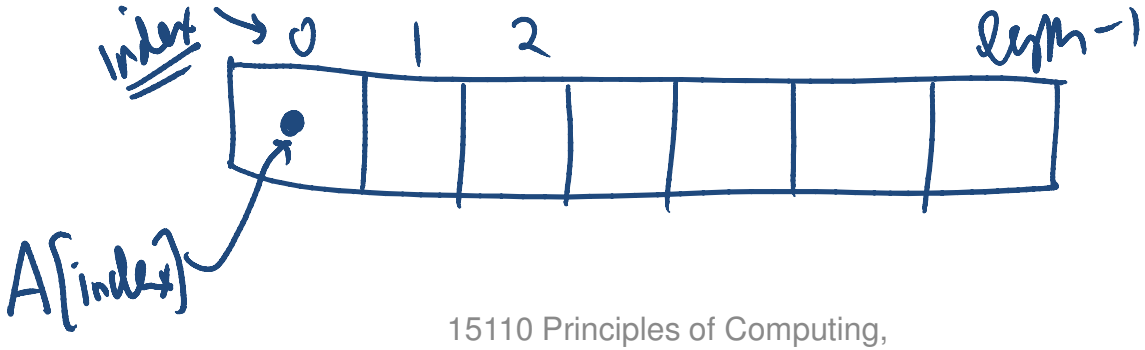
From devices to memory



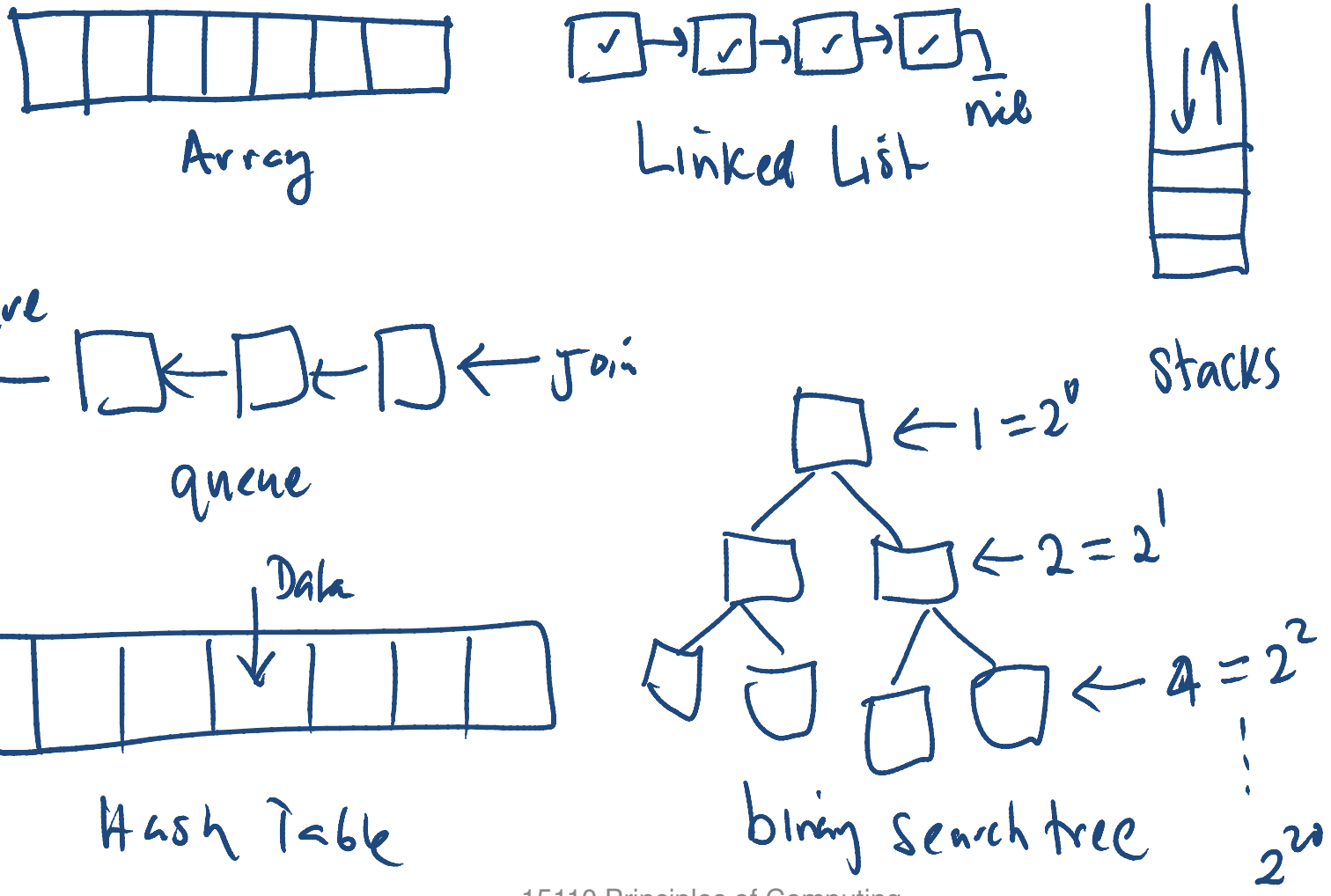
Data Structures

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.
- An **array** (or list) is a very simple data structure for holding a sequence of data.



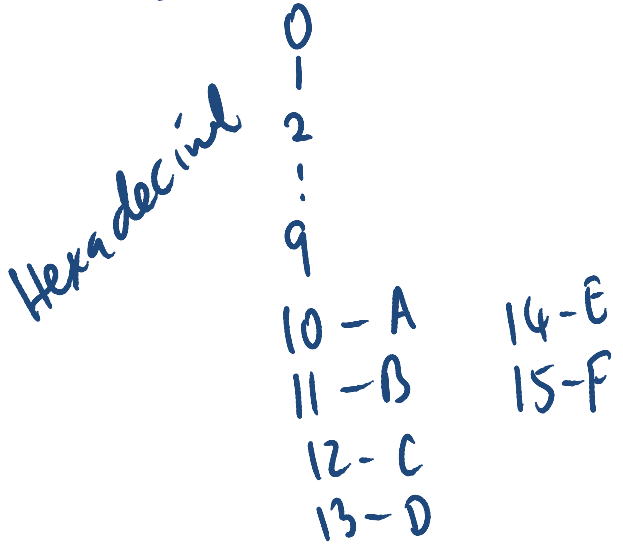
Examples of data structures

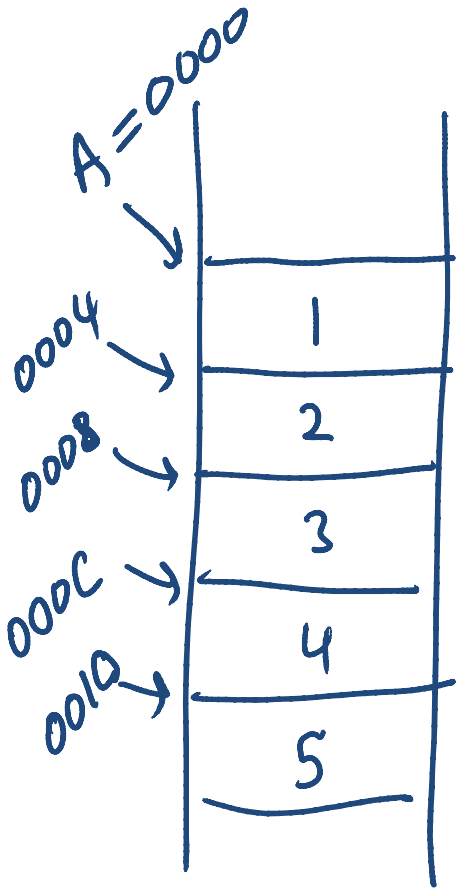


How Arrays are stored in memory

A = [1, 2, 3, 4, 5] ← 20 bytes

Integer — 4 bytes





Arrays: Pros and Cons

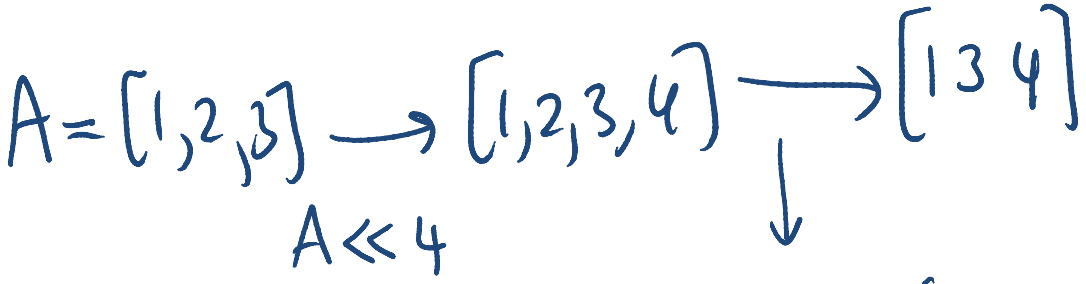
- Pros:
  - Access to an array element is fast since we can compute its location quickly.
- Cons:
  - If we want to insert or delete an element, we have to shift subsequent elements which slows our computation down.
  - We need a large enough block of memory to hold our array.

Array operations

- Finding length

A.length

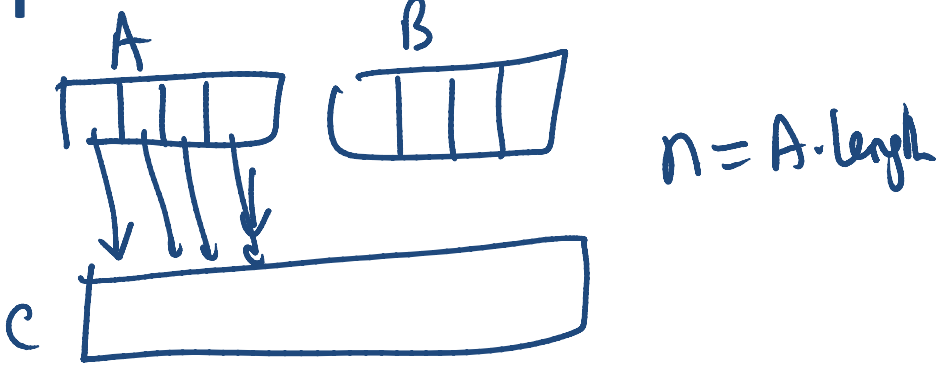
- Appending an element to end



- Removing an element
  - Array.delete_at(array.index(elem))

Array operations

- Merging two arrays



- Sorting an array

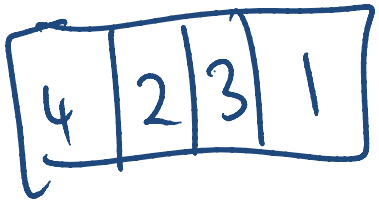
Insertion sort → $O(n^2)$

bubble sort, Selection Sort → $O(n^2)$

- Searching an array

Linear search, binary search

- Swapping two elements



tmp = A[0]
A[0] = A[n-1]
A[n-1] = tmp

$O(1)$

Merge two arrays

```
def   merge(list1, list2)

    n = list1.length
    m = list2.length
    C = [ ]
    for i from 0..n-1
       C<< list1[i]
    for j from 0..m-1
       C<< list2[j]
    return C

    end
```

Sub arrays

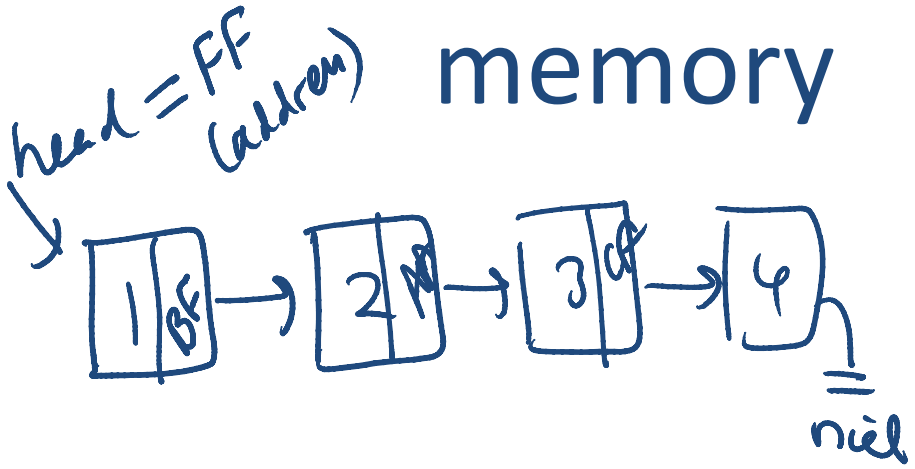
```
def  subarray(list, start, end)

    C=[ ]
    for i in start..end do
        C<<list[i]
    return C

end
```

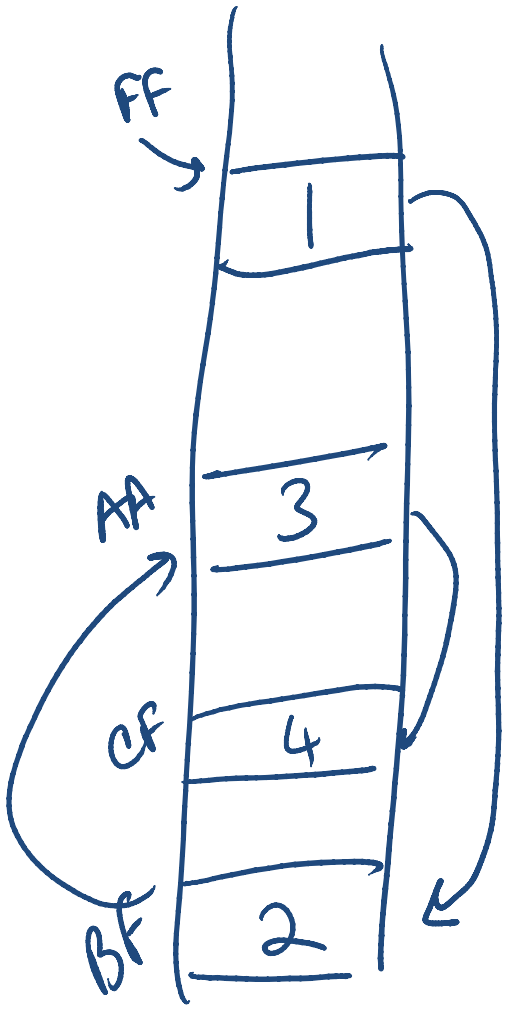
This returns a subarray from start to end

Linked Lists

- Another data structure that stores a sequence of data values is called 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.

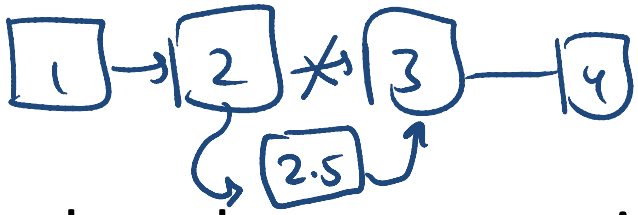
How Linked Lists are stored in memory





Linked List operations

Linked Lists: Pros and Cons



- Pros:
  - Inserting and deleting data does not require us to move/shift subsequent data elements.
    → flexible memory management
- 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 Arrays in Ruby

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

data[0]

data[2]

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

```
data[0]    =>  [1, 2, 3, 4]
data[1][2]  =>  7
data[2][5]  =>  nil
data[4][2]  =>  undefined method '[]' for nil
```

2D Array Example in Ruby

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

```
def sumMatrix(table)
  sum = 0
  for row in 0..table.length-1 do
    for col in 0..table[row].length-1 do
      sum = sum + table[row][col]
    end
  end
  return sum
end
```

number of rows in the table

number of columns in the given row of the table

Stacks

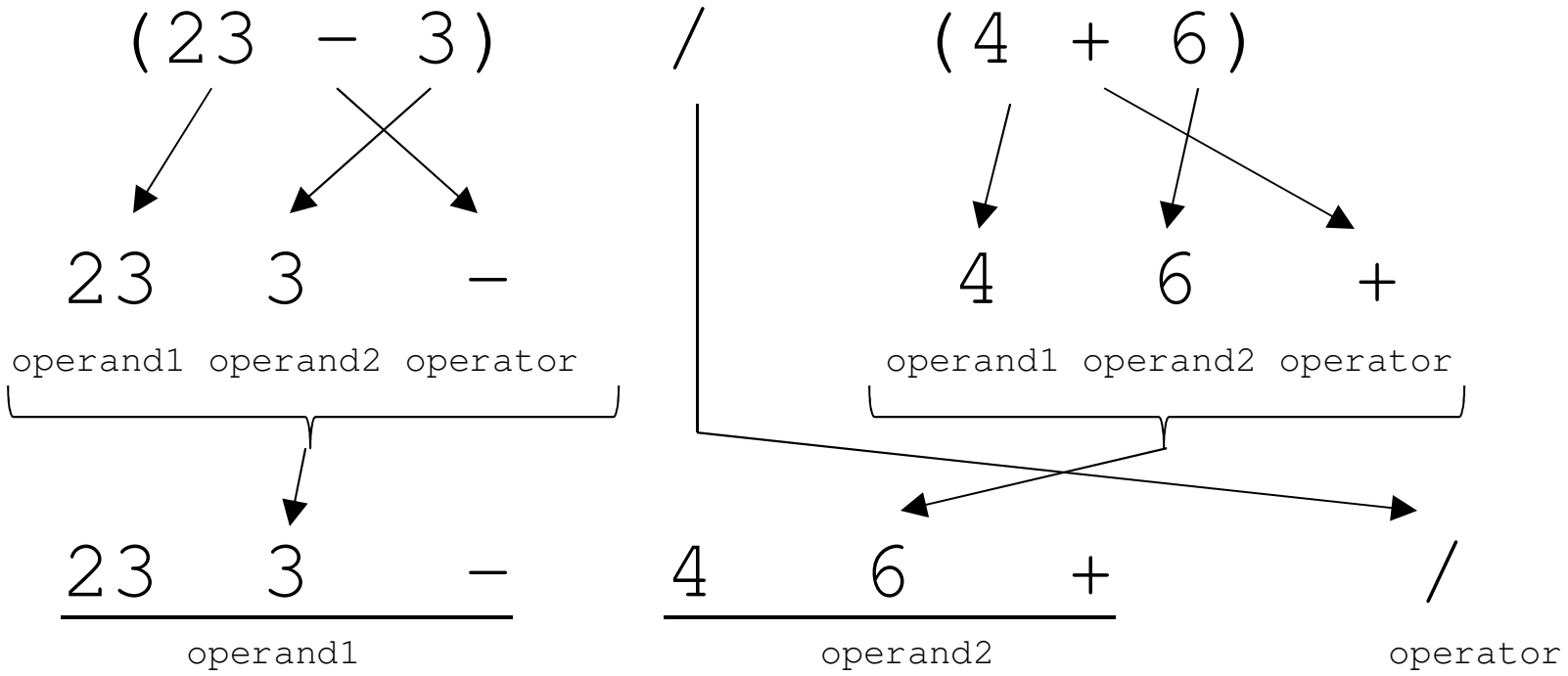
- A **stack** is a data structure that works on the principle of Last In First Out (LIFO).
  - LIFO: The last item put on the stack is the first item that can be taken off.
- Common stack operations:
  - Push – put a new element on to the top of the stack
  - Pop – remove the top element from the top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

RPN

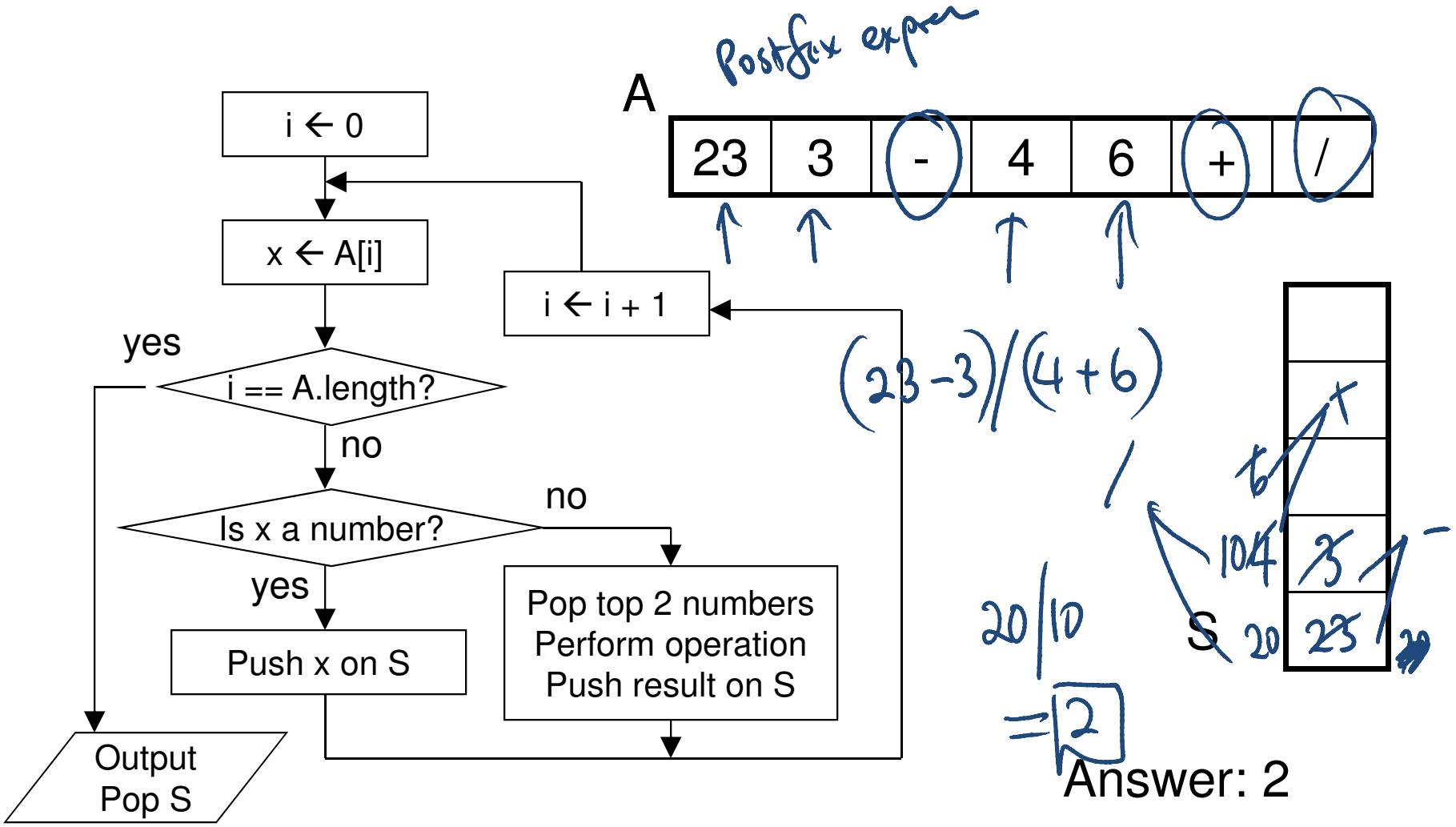
- Some modern calculators use Reverse Polish Notation (RPN)
  - Developed in 1920 by Jan Lukasiewicz
  - Computation of mathematical formulas can be done without using any parentheses
  - Example:
    (3 + 4) * 5 =
    becomes in RPN:
    3 4 + 5 *

RPN Example

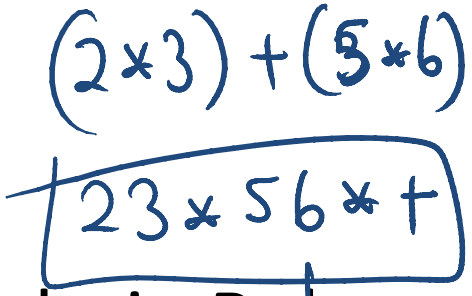
Convert the following standard mathematical expression into RPN:



Evaluating RPN with a Stack



Stacks in Ruby



- You can treat arrays (lists) as stacks in Ruby.

	stack	x
`stack = []`	`[]`	
`stack.push(1)`	`[1]`	
`stack.push(2)`	`[1,2]`	
`stack.push(3)`	`[1,2,3]`	
`x = stack.pop()`	`[1,2]`	`3`
`x = stack.pop()`	`[1]`	`2`
`x = stack.pop()`	`[]`	`1`
`x = stack.pop()`	`nil`	`nil`