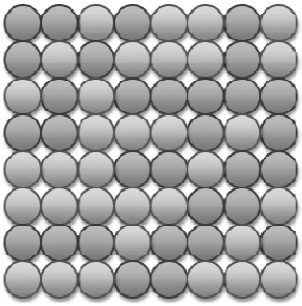


UNIT 6A
Organizing Data: Lists

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

Arrays in Memory

- Typically, array elements are stored in adjacent memory cells. The subscript (or index) is used to calculate an offset to find the desired element.
- Example: data = [50, 42, 85, 71, 99] Assume integers are stored using 4 bytes (32 bits).
- If we want data[3], we take the address of the start of the array and add the offset * the size of an array element to find the element we want.
- Do you see why the first index of an array is 0 now?

Address	Contents
100	50
104	42
108	85
112	71
116	99

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

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.

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: 50, 42, 85, 71, 99

Assume each integer and pointer requires 4 bytes.

Starting Location of List (head)
124

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

Linked List Example

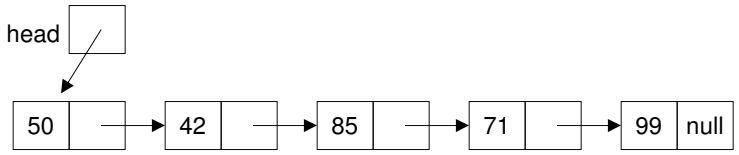
- To insert a new element, we only need to change a few pointers.
- Example: Insert 20 after 42.

Starting Location of List (head)
124

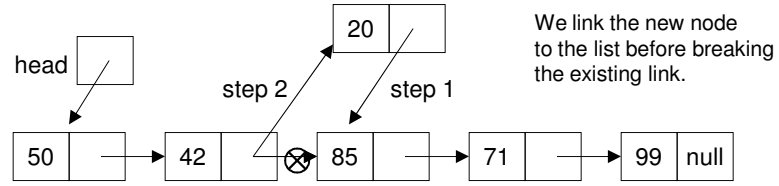
address	data	next
100	42	**156**
108	99	0 (null)
116		
124	50	100
132	71	108
140		
148	85	132
156	**20**	**148**

Drawing Linked Lists Abstractly

- L = [50, 42, 85, 71, 99]



- Inserting 20 after 42:



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 Arrays in Ruby

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

	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

Tracing the Nested Loop

```
for row in 0..table.length-1 do
  for col in 0..table[row].length-1 do
    sum = sum + table[row][col]
  end
end
```

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

```
table.length = 3
table[row].length = 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

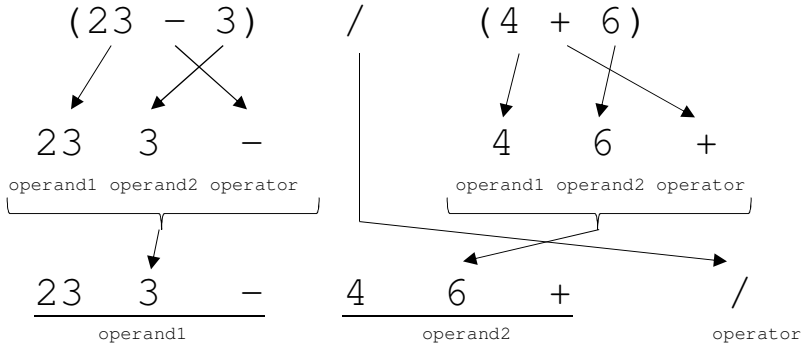
- A **stack** is a data structure that works on the principle of Last In First Out (LIFO).
  - LIFO: The last item put on the stack is the first item that can be taken off.
- Common stack operations:
  - Push – put a new element on to the top of the stack
  - Pop – remove the top element from the top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

RPN

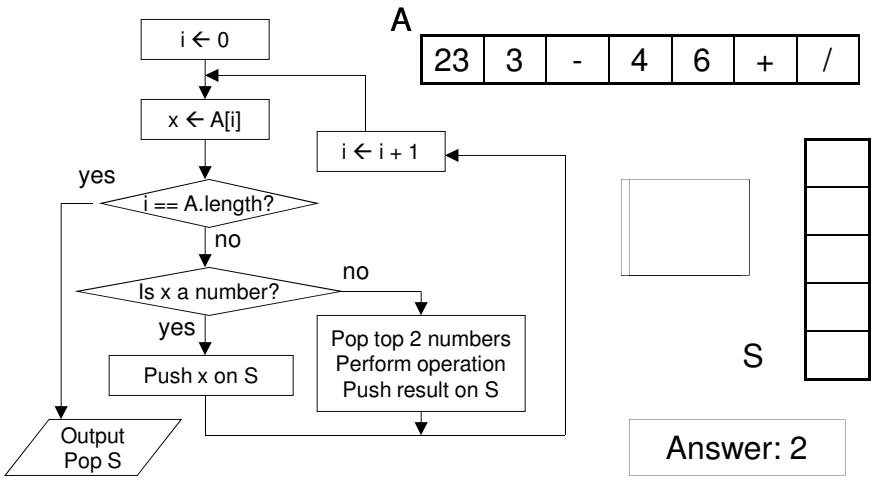
- Some modern calculators use Reverse Polish Notation (RPN)
  - Developed in 1920 by Jan Lukasiewicz
  - Computation of mathematical formulas can be done without using any parentheses
  - Example:
    (3 + 4) * 5 =
    becomes in RPN:
    3 4 + 5 *

RPN Example

Convert the following standard mathematical expression into RPN:

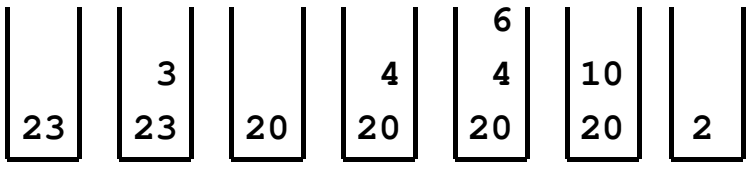


Evaluating RPN with a Stack



Example Step by Step

- RPN: `23 3 – 4 6 + /`
- Stack Trace:



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 stack 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

Josephus Problem

(from Wikipedia)

- There are people standing in a circle waiting to be executed. A certain number of people are skipped and then one person is executed. This is repeated starting with the person after the person who was executed. The elimination proceeds around the circle (which is becoming smaller as the executed people are removed) until one person remains.
  - Named after Flavius Josephus, a Jewish historian living in the 1st century. According to Josephus' account of the siege of Yodfat, he and his comrade soldiers were trapped in a cave, the exit of which was blocked by Romans. They chose suicide over capture and decided that they would form a circle and start killing themselves using a step of three. Josephus and another man remained alive last and gave up to the Romans.
- The goal is to determine where to stand in the circle so that you are the last person alive so you don't have to execute yourself.

Josephus Problem using Queues

Given a group of **n** men arranged in a circle such that every **m**th man will be executed until one remains. Find the position of the final survivor.

1. Fill Q with people numbered 1 through n.
2. While Q has more than 1 person do the following:
    a. Do the following m-1 times:
        i. Dequeue Q and store in x
        ii. Enqueue x on Q
    b. Then dequeue Q once (but don't enqueue)
3. Output the single person in Q as the last survivor.

Example: The Josephus Problem

Given a group of **n** men arranged in a circle such that every **m**th man will be executed until one remains, find the position of the last survivor.

1. Fill Q with people numbered 1 through n.
2. While Q has more than 1 person do the following:
    a. Do the following m-1 times:
        i. Dequeue Q and store in x
        ii. Enqueue x on Q
    b. Then dequeue Q once (but don't enqueue)
3. Output the single person in Q as the last survivor.

In this example, Josephus needs to be in position 1 to remain alive!

n = 6, m = 3

Queue Q

1 2 3 4 5 6

4 5 6 1 2

1 2 4 5

5 1 2

5 1

1