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], the computer takes the address of the start of the array and adds 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?

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>50</td>
</tr>
<tr>
<td>104</td>
<td>42</td>
</tr>
<tr>
<td>108</td>
<td>85</td>
</tr>
<tr>
<td>112</td>
<td>71</td>
</tr>
<tr>
<td>116</td>
<td>99</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>address</th>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>42</td>
<td>148</td>
</tr>
<tr>
<td>108</td>
<td>99</td>
<td>0 (null)</td>
</tr>
<tr>
<td>116</td>
<td></td>
<td></td>
</tr>
<tr>
<td>124</td>
<td>50</td>
<td>100</td>
</tr>
<tr>
<td>132</td>
<td>71</td>
<td>108</td>
</tr>
<tr>
<td>140</td>
<td></td>
<td></td>
</tr>
<tr>
<td>148</td>
<td>85</td>
<td>132</td>
</tr>
<tr>
<td>156</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assume each integer and pointer requires 4 bytes.

Starting Location of List (head) 124
Linked List Example

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

<table>
<thead>
<tr>
<th>address</th>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>42</td>
<td>156</td>
</tr>
<tr>
<td>108</td>
<td>99</td>
<td>0 (null)</td>
</tr>
<tr>
<td>116</td>
<td></td>
<td></td>
</tr>
<tr>
<td>124</td>
<td>50</td>
<td>100</td>
</tr>
<tr>
<td>132</td>
<td>71</td>
<td>108</td>
</tr>
<tr>
<td>140</td>
<td></td>
<td></td>
</tr>
<tr>
<td>148</td>
<td>85</td>
<td>132</td>
</tr>
<tr>
<td>156</td>
<td>20</td>
<td>148</td>
</tr>
</tbody>
</table>

Starting Location of List (head)

124

Drawing Linked Lists Abstractly

- \( L = [50, 42, 85, 71, 99] \)

- Inserting 20 after 42:

We link the new node to the list before breaking the existing link.
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
```

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>18</td>
<td>43</td>
<td>49</td>
<td>65</td>
</tr>
<tr>
<td>1</td>
<td>14</td>
<td>30</td>
<td>32</td>
<td>53</td>
<td>75</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>28</td>
<td>38</td>
<td>50</td>
<td>73</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>24</td>
<td>37</td>
<td>58</td>
<td>62</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>19</td>
<td>40</td>
<td>46</td>
<td>66</td>
</tr>
</tbody>
</table>
2D Arrays in Ruby

```ruby
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]` => `nil`
- `data[4][2]` => `undefined method '[]' for nil`

```
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

```ruby
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
```

```
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
```
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
```

Table:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

`table.length = 3`
`table[row].length = 4` for every row

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:
    \((3 + 4) \times 5 =\)
    becomes in RPN:
    \(3 \ 4 \ + \ 5 \ *

RPN Example

Convert the following standard mathematical expression into RPN:

\[
\begin{align*}
(23 - 3) & \div (4 + 6) \\
23 & \ 3 \ - \\
4 & \ 6 \ + \\
23 & \ 3 \ - \\
4 & \ 6 \ + \\
\end{align*}
\]
Evaluating RPN with a Stack

Example Step by Step

- **RPN:** \[23 \ 3 \ - \ 4 \ 6 \ + \ /\]

- **Stack Trace:**

  23  3  20  4  6  4  10  20  2
Stacks in Ruby

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

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

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