UNIT 6A
Organizing Data: Lists

Last Two Weeks

- Algorithms: Searching and sorting
- Problem solving technique: Recursion
- Asymptotic worst-case analysis using the big O notation
This Week

• Observe: The choice of how to organize your data determines how efficiently you can do certain types of computation

• Data structures
  – Arrays, linked lists, stacks, queues, hash tables, trees, graphs ...

• Representation of dynamic sets and supporting operations such as search, insert, delete ...

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], the computer takes the address of the start of the array and adds the offset \( \times \) 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>

Starting Location of List (head) 124

Assume each integer and pointer requires 4 bytes.
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 =  
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
```

\[ B[2][3] = 50 \]
2D Arrays in 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`

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
```
Tracing the Nested Loop

```plaintext
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>
<thead>
<tr>
<th>row</th>
<th>col</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>21</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>28</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>36</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>45</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>55</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>66</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>78</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:
    \[3 \times 4 + 5 = \]

RPN Example

Convert the following standard mathematical expression into RPN:

\[
\frac{23 - 3}{4 + 6}
\]

In RPN:

\[
23 \ 3 \ - \ 4 \ 6 \ + \ /
\]
Evaluating RPN with a Stack

A

23 3 - 4 6 + /

i ← 0
x ← A[i]

yes

Is x a number?

yes

Push x on S

no

i ← i + 1

no

return

S

Output

Pop S

Example Step by Step

• RPN: 23 3 - 4 6 + /

• Stack Trace:

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>3</td>
<td>20</td>
<td>4</td>
<td>6</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>2</td>
</tr>
</tbody>
</table>

Answer: 2
Stacks in Ruby

- You can treat arrays (lists) as stacks in 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
Comparing Algorithms

- You are a professor and you want to find an exam in a large pile of n exams.
- Search the pile using linear search.
  - Per student: $O(n)$
  - Total for n students: $O(n^2)$
- Have an assistant sort the exams first by last name.
  - Assistant’s work: $O(n \log n)$ using merge sort
  - Professor:
    - Search for one student: $O(\log n)$ using binary search
    - Total for n students: $O(n \log n)$
Another way

• Set up a large number of “buckets”.
• Place each exam into a bucket based on some function.
  – Example: 100 buckets, each labeled with a value from 00 to 99. Use the student’s last two digits of their student ID number to choose the bucket.
• Ideally, if the exams get distributed evenly, there will be only a few exams per bucket.
  – Assistant: $O(n)$ putting $n$ exams into the buckets
  – Professor: $O(1)$ search for an exam by going directly to the relevant bucket and searching through a few exams.

Hashing

A hash function $h$ is used to map keys to hash-table slots
Strings and ASCII codes

```ruby
s = "hello"
for i in 0..s.length-1 do
  print s[i], "\n"
end
```

104 101 108 108 111

You can treat a string like an array in Ruby.
If you access the ith character, you get the ASCII code for that character.

Hash table

- Let’s assume that we are going to store only lower case strings into an array (hash table).

```ruby
table1 = Array.new(26)
=> [nil, nil, nil, nil, nil, nil, nil, nil, nil, nil, nil, nil, nil, nil, nil, nil, nil, nil, nil, nil, nil, nil, nil, nil, nil]
```
Hash table

• We could pick the array position where each string is stored based on the first letter of the string using this hash function:

```python
def h(string):
    return string[0] - 97
end
```

The ASCII values of lowercase letters are:
“a” -> 97, “b” -> 98, “c” -> 99, “d” -> 100, etc.

Inserting into Hash Table

• To insert into the hash table, we simply use the hash function h to determine which index (“bucket”) to store the element.

```python
def insert(table, name):
    table[h(name)] = name
end

insert(table1, “aardvark”)
insert(table1, “beaver”) ...
```
Hash function (cont’d)

• Using the hash function $h$:
  - “aardvark” would be stored in an array at index 0
  - “beaver” would be stored in an array at index 1
  - “kangaroo” would be stored in an array at index 10
  - “whale” would be stored in an array at index 22

```plaintext
table1 => ["aardvark", "beaver", nil, nil, nil, nil, nil, nil, nil, nil, "kangaroo", nil, nil, nil, nil, nil, nil, nil, nil, nil, "whale", nil, nil, nil]
```

Hash function (cont’d)

• But if we try to insert “bunny” and “bear” into the hash table, each word overwrites the previous word since they all hash to index 1:

```plaintext
>> insert(table1,"bunny")
>> insert(table1,"bear")
>> table1
=> ["aardvark", "bear", nil, nil, nil, nil, nil, nil, nil, nil, nil, "kangaroo", nil, nil, nil, nil, nil, nil, nil, nil, nil, nil, nil, nil, "whale", nil, nil, nil]
```
Revised Hash table

- Let’s make our hash table an array of arrays (an array of “buckets”)
- Each bucket can hold more than one string.

```ruby
table2 = Array.new(26)
for i in 0..25 do
    table2[i] = []
end
=> [[]], [[]], [[]], [[]], [[]], [[]], [[]], [[]], [[]], [[]], [[]], [[]], [[]], [[]], [[]], [[]], [[]], [[]], [[]], [[]], [[]], [[]], [[]], [[]]"
```

Revised insert function

```ruby
def insert(table, key)
    # find the bucket (array) in the table
    # array using the hash function h
    bucket = table[h(key)]
    # append the key string to the bucket
    # array
    bucket << key
end
```
Inserting into new hash table

```python
insert(table2, "aardvark")
>> insert(table2, "beaver")
>> insert(table2, "kangaroo")
>> insert(table2, "whale")
>> insert(table2, "bunny")
>> insert(table2, "bear")
>> table2
=> [["aardvark"], ["beaver", "bunny", "bear"], [], [], [], [], [], [], [], [], ["kangaroo"], [], [], [], [], [], [], [], [], [], [], [], ["whale"], [], [], []]
```

Collisions

- “beaver”, “bunny” and “bear” all end up in the same bucket.
- These are collisions in a hash table.
- The more collisions you have in a bucket, the more you have to search in the bucket to find the desired element.
- We want to try to minimize the collisions by creating a hash function that distribute the keys (strings) into different buckets as evenly as possible.
First Try

def h(string):
    k = 0
    for i in range(string.length-1):
        k = string[i] + k
    return k

h("hello") => 532
h("olleh") => 532
Permutations still give same index (collision) and numbers are high.

Second Try

def h(string):
    k = 0
    for i in range(string.length-1):
        k = string[i] + k*256
    return k

h("hello") => 448378203247
h("olleh") => 478560413032
Better, but numbers are still high. We probably don’t want to (or can’t) create arrays that have indices this large.
Third Try

def h(string, tablesize)
    k = 0
    for i in 0..string.length-1 do
        k = string[i] + k*256
    end
    return k % tablesize
end

We can use the modulo operator to take the large values and map them to indices for a smaller array.

Revised insert function

def insert(table, key)
    # find the bucket (array) in the table
    # array using the hash function h
    bucket = table[h(key, table.length)]
    # append the key string to the bucket
    # array
    bucket << key
end
Final results

table3 = Array.new(13)
for i in 0..12 do table3[i] = [] end
=> [[]], [[]], [[]], [[]], [[]], [[]], [[]], [[]], [[]], [[]], [[]], [[]], [[]]
>> insert(table3,"aardvark")
>> insert(table3,"bear")
>> insert(table3,"bunny")
>> insert(table3,"beaver")
>> insert(table3,"dog")
>> table3
=> [[]], [[]], [[]], [[]], [[]], [[]], [[]], [[]], [["bunny"],
["aardvark", "bear"], ["dog"], ["beaver"]]

Still have one collision, but b-words are distributed better.

Searching in a hash table

To search for a key, use the hash function to find out which bucket it should be in, if it is in the table at all.

def contains?(table, key)
  bucket = table[h(key,table.length)]
  for entry in bucket do
    return true if entry == key
  end
  return false
end
Efficiency

• If the keys (strings) are distributed well throughout the table, then each bucket will only have a few keys and the search should take $O(1)$ time.

• Example:
  If we have a table of size 1000 and we hash 4000 keys into the table and each bucket has approximately the same number of keys (approx. 4), then a search will only require us to look at approx. 4 keys $=> O(1)$
  – But, the distribution of keys is dependent on the keys and the hash function we use!