3-Stack-Queue and Graphs

Bruno MARTIN,
University of Nice - Sophia Antipolis
mailto:Bruno.Martin@unice.fr
http://deptinfo.unice.fr/~bmartin/mathmods.html

Stack implementation

Basic operations:

- push: adds a element on the top of the stack
- pop: removes and returns the top element

already in ruby

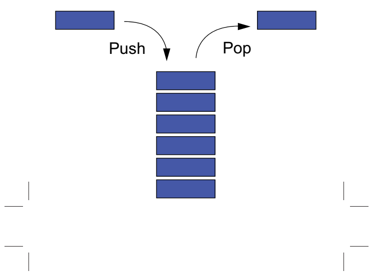
```
stack = Stack.new
stack.push(3)
stack.push(100)
stack.count
stack.pop()
```

```
class Stack
  def initialize
   @the_stack = []
  end

  def push(item)
    @the_stack.push item
  end

  def pop
    @the_stack.pop
  end

  def count
    @the_stack.length
  end
end
```


Stack data structure

Based on the LIFO principle.
Used for removing recursive calls
Basic operations:

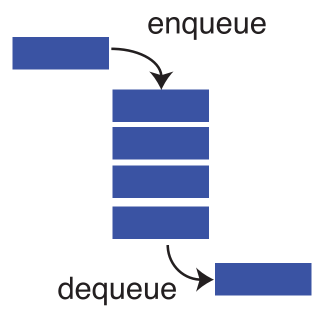
- push: adds a element on the top of the stack
- pop: removes and returns the top element



Queue data structure

Based on the FIFO principle.
Used for tree/graph traversal
Basic operations:

- enqueue: adds a element on the top of the queue
- dequeue: removes and returns the bottom element



Stack implementation

Basic operations:

- enqueue: adds a element on the top of the queue
- dequeue: removes and returns the bottom element

already in ruby

```
queue = Queue.new
queue.enqueue(2)
queue.enqueue(3)
queue.dequeue
```

```
class Queue
  def initialize
   @the_queue = []
  end
  def enqueue(item)
    @the_queue.push item
  end
  def dequeue
    @the_queue.shift
  end
  def count
    @the_queue.length
  end
  def empty?
    return @the_queue.length
  end
end
```


Basics definitions of Graphs

- A **graph** $G = (V, E)$ is a collection of vertices V and edges E
- An **edge** is a pair of vertices (s, t). And t is **adjacent** to s
- A **path** from v_1 to v_n is a list of vertices v_1, v_2,...,v_n so that successive vertices are connected by edges
- A **simple path** is a path in which no vertex is repeated
- A **cycle** is a path where the first and the last vertex are the same

Graphs

Many problems are naturally formulated in terms of objects and relationships among them : **airline route map**, **electric circuits**, **job scheduling**,... **Graphs** model such situations
On such different types of graphs, we address different questions:

"Which is the fastest (cheapest) way to get from one city to another?"

The Shortest Paths Problem

"Is every element of an electric circuit connected with the others?"

The Connectivity Problem

"When should each task be performed ?"

The Topological Sorting

Directed or Non-directed Graphs ?

A **graph** is **directed** (a digraph) when the pair of vertices is ordered $s \to t$, s is the **source** and t is the **target**
Some concepts are intuitively better defined on **digraphs** some others on **non-directed graphs**
All the concepts may be applied on **any graphs** provided that you make the **appropriate transformation**:

- You **transform** a **digraph** into a **non-directed graph** by **removing** the **orientation** of the **edges**
- You **transform** a **non-directed graph** into a **digraph** by considering **two directed** edges for **each non-directed** edge

Basics definitions of Graphs

A graph is **connected** if there is a non-directed path from every vertex to every other vertex in the graph

A directed graph is **strongly connected** if there is a directed path from every vertex to every other vertex in the graph

A **spanning tree** of a graph is the subgraph that contains all the vertices but only enough of the edges to form a tree

We can attach informations to the vertices and the edges of a graph. An information can be a label (**labeled graph**) or a value of any given data type (**valuated graph**)

A graph with all edges present is **complete**

Representing a Graph with adjacency lists

The **adjacency list** of vertex i is the list of all adjacent vertices

G is an array of V elements where $G[i]$ is a pointer to the adjacency list of the vertex i

Advantage: It requires a **storage** proportional to $V + E$

Drawback: It needs at most $O(E)$ time to determine whether there is an edge from vertex i to j

The appropriate choice of data depends on the operations that will be applied to the vertices and edges of the graph

Representing a Graph with an Adjacency Matrix

The **adjacency matrix** A is a matrix $V \times V$ of booleans (resp. label) where $A[i][j]$ is $TRUE$ (reps. a legal label) if there is an edge from vertex i to j

Advantage:

- The time required to **access an element** of an adjacency matrix is independent of the size V and E: **constant time**
- Adjacency Matrix representation for graph is choosen **for algorithm** which **frequently** need to know whether a **given edge is present**

Drawback:

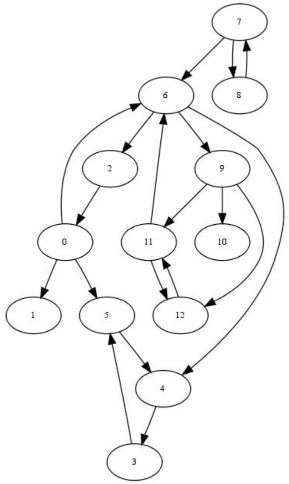
- It requires V^2 **storage** even if the graph is sparse
- Read or examine the matrix would require $O(V^2)$ time which would **preclude** $O(E)$ algorithms for manipulating graphs with E edges

The graph G

$$\text{A graph } G = (\{A, B, C, D, E, F, G, H, I, J, K, L, M\}, \{(A, F), (A, B), (A, G), (C, A), (D, F), (E, D), (F, E), (G, C), (G, E), (G, J), (H, G), (H, I), (I, H), (J, K), (J, L), (J, M), (L, G), (L, M), (M, L)\})$$

The graph G

```
gem install rgl
 irb
>> require 'rgl/adjacency'
>> GG=RGL::DirectedAdjacencyGraph[
0,5 ,0,1 ,0,6 ,2,0 ,3,5 ,4,3
,5,4 ,6,2 ,6,4 ,6,9 ,7,6 ,7,8
,8,7 ,9,10 ,9,11 ,9,12 ,11,6
,11,12 ,12,11]
>>require 'rgl/dot'
>> GG.write_to_graphic_file('jpg')
```



The Adjacency List of the graph G

A → B → F → G → □
B → □
C → A → □
D → F → □
E → D → □
F → E → □
G → C → E → J → □
H → G → I → □
I → H → □
J → K → L → M → □
K → □
L → G → M → □
M → L → □

The Adjacency Matrix of the graph G

	A 0	B 1	C 2	D 3	E 4	F 5	G 6	H 7	I 8	J 9	K 10	L 11	M 12
A 0	0	0	1	0	0	0	0	0	0	0	0	0	0
B 1	1	0	0	0	0	0	0	0	0	0	0	0	0
C 2	0	0	0	0	0	0	1	0	0	0	0	0	0
D 3	0	0	0	0	1	0	0	0	0	0	0	0	0
E 4	0	0	0	0	0	1	1	0	0	0	0	0	0
F 5	1	0	0	1	0	0	0	0	0	0	0	0	0
G 6	1	0	0	0	0	0	0	1	0	0	0	1	0
H 7	0	0	0	0	0	0	0	0	1	0	0	0	0
I 8	0	0	0	0	0	0	0	1	0	0	0	0	0
J 9	0	0	0	0	0	0	1	0	0	0	0	0	0
K 10	0	0	0	0	0	0	0	0	0	1	0	0	0
L 11	0	0	0	0	0	0	0	0	0	1	0	0	1
M 12	0	0	0	0	0	0	0	0	0	1	0	1	0

If there is an edge from i (horizontal) to j (vertical) then set $M[i][j]$ to 1 else set it to 0

Graph traversals

Problem: How to **traverse** the graph i.e. systematically visit every vertices?

As for trees, 2 ways to proceed. Start on an initial vertex, (**root**):

DFS Depth first search: starts at the root and explores as far as possible along each branch before backtracking. Much like preorder traversal of a tree

BFS Breadth first search: starts at the root and explores all the neighboring nodes. Then for each of those nearest nodes, it explores their unexplored neighbor nodes, and so forth. Much like level-order traversal of a tree

The Depth-First Search Algorithm

Problem Find a natural way to systematically visit every vertex and every edge of a directed graph :

- Start from one vertex
- Step forward all along one path (without passing through a vertex already visited)
- When you are stuck, turn back until you can step forward an unvisited vertex
- Recursivity offers you the backtrack for free

Depth-First Search

```
dfs(v)
    visit(v)
    for each neighbor w of v
        if w is unvisited
           dfs(w)
    #      add edge vw to tree T
        end
```


The Depth-first Search algorithm

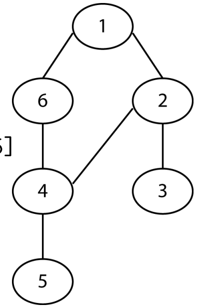
- **Initially** mark all vertices as unvisited
- Select one vertex *v* in *G* as the start vertex
- Mark *v* as being visited
- Run Depth-First Search recursively on each unvisited vertex adjacent to *v*
- Once all vertices that can be reached from *v* have been visited the Depth-First Search of *v* terminates
- If some vertices remain unvisited, select one of them as a new start vertex
- Repeat this process until all vertices have been visited

Ruby implementation

```
require 'rgl/adjacency'
class Graphe < RGL::AdjacencyGraph
  def dfs
    $visited = Array.new(G.max+1)
    G.each_vertex{ |i| $visited[i]=false }
    def mydfs(n)
      $visited[n] = true
      puts n
      G.each_adjacent(n){ |x|
        mydfs(x) if $visited[x]==false }
    end
    puts "from which node?"
    v=gets
    G.mydfs(v.to_i)
    end
end
```

Usage

```
>> require "dfs.rb"
>> G=Graphe[1,2 ,2,3 ,1,6 ,6,4 ,2,4 ,4,5]
>> G.dfs
2 1 6 4 5 3
```



Iterative Depth-First Search

DFS invoked on a graph is exactly equivalent to traversing a tree that spans the graph we call it **tree traversal**
The recursion of DFS can be removed by using a stack
A vertex can be unvisited, unvisited and in the stack, or visited, in this case it is not in the stack
We must avoid putting a vertex twice on the stack

Running time of DFS

The graph has E edges and V vertices

Adjacency list All the calls to DFS take $O(V+E)$ time
- DFS is called once by vertex $O(V)$
- Going down the adjacency list of all vertices is proportional to the sums of the lengths of those lists i.e. $O(E)$

Adjacency matrix DFS takes $O(V^2)$ time
- DFS is called once by vertex V
- Going down the adjacency list of one vertex costs exactly V and we do it for each vertex so $O(V^2)$

Iterative DFS

```
dfs(s)
  initialize S to be a stack with one element s
  while S not empty
    take a node u from S
    if explored[u]=false then
       set explored[u]= true
       for each edge (u,v) adjacent to u
          add v to S
       end
    end
  end
```

Execution seen at http://www.cs.umd.edu/class/sum2005/cmsc451/dfsimplementation.pdf

The Breadth-First Search algorithm

Problem: It is another systematic way of visiting the vertices of a digraph $G(V, E)$. Start from a vertex, step forward all vertices adjacent to it, then step forward all vertices adjacent to its sons,...

The Breadth-First Search algorithm is quite the same algorithm as the iterative DFS, you simply replace the stack with a queue

Ruby implementation of BFS

```
def bfs
    $explored = Array.new(G.max+1)
    G.each_vertex{ |i| $explored[i]=false }
    def bfs_from(s)
      q=Queue.new
      q.enqueue s
      until q.empty?
        u=q.dequeue
        if not $explored[u]
          $explored[u]=true
          puts u
          G.each_adjacent(u) { |v| q.enqueue v}
        end
      end
    end
    v = gets
    G.bfs_from(v.to_i)
  end
```


(Iterative) BFS

```
bfs(s)
  initialize Q to be a queue with one element s
  while Q not empty
    take a node u from Q
    if explored[u]=false then
       set explored[u]= true
       for each edge (u,v) adjacent to u
          add v to Q
       end
    end
  end
```