# Documentation

## Problem statement

18. Path in a maze. Consider a maze made of occupied (X) and empty (\*) cells. Let R be a

robot in this maze.

Requirements:

a) Test if R can get out of the maze.

b) Determine a path to get out of the maze (if there is one).

c) Determine a path with minimum length to get out of the maze.

ADTs to be used: Stack (implemented on a singly linked list on an array) and Queue

(implemented on a doubly linked list on an array) .

***Justification***

I used Stack and Queue to solve this problem because I needed the queue to keep the elements in order to use BFS algorithm and the stack to keep the directions for the robot. If there is a path in one direction, the coordonates of the node will be pushed in the queue in order to acces all the adiacent nodes from that node.

## ADT Stack specification and interface

The ADT Stack represents a container in which access to the elements is restricted to one end of the container, called the top of the stack.

When a new element is added, it will automatically be added at the top.

When an element is removed it will be removed automatically from the top.

Only the element from the top can be accessed.

Because of this restricted access, the stack is said to have a LIFO policy: Last In, First Out (the last element that was added will be the first element that will be removed).

**Note**: stacks cannot be iterated, so they don’t have an iterator operation!

init(s)

* Description: creates a new empty stack
* Pre: True
* Post: s ∈ S, s is an empty stack

destroy(s)

* Description: destroys a stack
* Pre: s ∈ S
* Post: s was destroyed

push(s, e)

* Description: pushes (adds) a new element onto the stack
* Pre: s ∈ S, e is a TElem
* Post: s’ ∈ S, s’ = s ⊕ e, e is the most recent element added to the stack

pop(s)

* Description: pops (removes) the most recent element from the stack
* Pre: s ∈ S
* Post: pop ← e, e is a TElem, e is the most recent element from s, s’ ∈ S, s’ = s Θ e
* Throws: an underflow error if the stack is empty

top(s)

* Description: returns the most recent element from the stack (but it does not change the stack)
* Pre: s ∈ S
* Post: top ← e, e is a TElem, e is the most recent element from s
* Throws: an underflow error if the stack is empty

isEmpty(s)

* Description: checks if the stack is empty (has no elements)
* Pre: s ∈ S
* Post: i

isFull(s)

* Description: checks if the stack is full
* Pre: s ∈ S
* Post: i𝑠𝐹𝑢𝑙𝑙<−{𝑡𝑟𝑢𝑒− 𝑖𝑓 𝑠 𝑖𝑠 𝑓𝑢𝑙𝑙𝑓𝑎𝑙𝑠𝑒−𝑜𝑡ℎ𝑒𝑟𝑤𝑖𝑠𝑒

## ADT Stack representation and implementation on SLLA

SLLA:

* elems: TElem[]
* next: Integer[]
* head: Integer
* firstEmpty: Integer
* capacity : integer

***Node Implementation***

StackNode:

Integer line

Integer column

Interger next

Function getLine(n):

getLine<- n.line complexity theta(1)

end

Function getColumn(n):

getColumn<-n.column complexity theta(1)

end

function getNext(n):

getNext<-n.next complexity theta(1)

end

function setNext(n,next):

n.next<-next Complexity theta(1)

end

***Stack Implementation***

Stack:

StackNode elems[capcity]

integer capacity

int top

int firstEmpty

function init(s):

s.cap = CAPACITY

s.top = -1

s.firstEmpty = 0

for i <- 0, s.cap – 1 Complexity O(n)

s.elems[i].setNext(i + 1)

s.elems[s.cap - 1].setNext(-1)

end

function push(s,StackNode e):

if s.isFull()=true

throw Exception "Stack is full"

pos <- s.firstEmpty

s.firstEmpty <- s.elems[pos].getNext() Complexity theta(1)

s.elems[pos] <- e

s.elems[pos].setNext(s.top)

s.top <- pos

end

function pop(s):

if s.isEmpty()=true

throw Exception "Stack is empty"

current <-s.top

topElem <- s.elems[s.top]

s.top <- s.elems[s.top].getNext() Complexity theta(1)

s.elems[current].set\_next(s.firstEmpty)

s.firstEmpty <- current

pop<- topElem

end

function topElement(s):

if s.top = -1

topElement<- StackNode() Complexity theta(1)

topElement <- elems[s.top]

end

function isFull(s):

if s.cap-1 = s.top

isFull <- true Complexity theta(1)

isFull <- false

end

function isEmpty(s):

if s.top = -1

isEmpty <- true Complexity theta(1)

isEmpty <- false

end

## ADT Queue specification and interface

The ADT Queue represents a container in which access to the elements is restricted to the two ends of the container, called front and rear.

When a new element is added (pushed), it has to be added to the rear of the queue.

When an element is removed (popped), it will be the one at the front of the queue.

Because of this restricted access, the queue is said to have a FIFO policy: First In First Out.

The domain of the ADT Queue: Q = {q|q is a queue with elements of type TElem}

**Note**: queues cannot be iterated, so they don’t have an iterator operation!

init(q)

* Description: creates a new empty queue
* Pre: True
* Post: q ∈ Q, q is an empty queue

destroy(q)

* Description: destroys a queue
* Pre: q ∈ Q
* Post: q was destroyed

push(q, e)

* Description: pushes (adds) a new element to the rear of thequeue
* Pre: q ∈ Q, e is a TElem
* Post: q’ ∈ Q, q’ = q ⊕ e, e is the element at the rear of the queue

pop(q)

* Description: pops (removes) the element from the front ofthe queue
* Pre: q ∈ Q
* Post: pop ← e, e is a TElem, e is the element at the front of q, q’ ∈ Q, q’ = q Θ e
* Throws: an underflow error if the queue is empty

top(q)

* Description: returns the element from the front of the queue Pre: q ∈ Q
* Post: top ← e, e is a TElem, e is the element from the front of q
* Throws: an underflow error if the queue is empty

isEmpty(s)

* Description: checks if the queue is empty (has no elements)
* Pre: q ∈ Q
* Post: i

isFull(s)

* Description: checks if the stack is full
* Pre: q ∈ Q
* Post: i

## ADT Queue representation and implementation on DLLA

In theory, we have two options:

Put front at the beginning of the list and rear at the end

Put front at the end of the list and rear at the beginning

In either case we will have both operations (push or pop) in Θ(1) complexity.

DLLANode:

info: TElem

next: Integer

prev: Integer

DLLA:

nodes: DLLANode[]

head: Integer

tail: Integer

firstEmpty: Integer

capacity : integer

***QueueNode Implementation***

QueueNode:

integer line

integer col

integer next

integer prev

function set\_next(n,next)

n.next <- next Complexity theta(1)

end

function set\_prev(n,prev)

n.prev <- prev Complexity theta(1)

end

function getLine(n)

getLine <-line Complexity theta(1)

end

function getColumn(n)

getColumn <- n.col Complexity theta(1)

end

function getNext(n)

getNext <- n.next Complexity theta(1)

end

function getPrev(n)

getPrev <- n.prev Complexity theta(1)

end

***Queue Implementation***

Queue:

QueueNode elems[CAPACITY]

integer cap

integer front

integer rear

integer firstEmpty

function init(q)

q.cap = CAPACITY

q.front = -1

q.rear = -1

q.firstEmpty = 0

for i <- 0, q.cap – 1 Complexity O(n)

q.elems[i].set\_prev(i - 1)

q.elems[i].set\_next(i + 1)

q.elems[q.cap - 1].set\_prev(q.cap - 2)

q.elems[q.cap - 1].set\_next(-1)

end

function push(q,e)

if q.isFull()

throw Exception"Queue full"

pos <- q.firstEmpty

q.firstEmpty <- q.elems[pos].get\_next() Complexity theta(1)

q.elems[q.firstEmpty].set\_prev(-1)

q.elems[pos] <- e

if q.front = -1

q.front <- pos

q.rear <- pos

q.elems[pos].set\_next(-1)

else

q.elems[q.rear].set\_next(pos)

q.elems[pos].set\_prev(q.rear)

q.elems[pos].set\_next(-1)

q.rear <- pos

end

function pop(q):

QueueNode frontElem;

if q.isEmpty()

throw Exception "Queue is empty"

else

frontElem <- q.elems[this->front]

if q.front = q.rear

q.elems[this->front].set\_next(q.firstEmpty)

q.elems[q.firstEmpty].set\_prev(q.front)

q.firstEmpty <- q.front

q.front <- -1

q.rear <- -1

else

q.elems[q.firstEmpty].set\_prev(q.front)

pos <- q.front

q.front <- q.elems[q.front].get\_next()

q.elems[q.front].set\_prev(-1)

q.elems[pos].set\_next(q.firstEmpty)

q.firstEmpty = pos

pop <- frontElem

end

Complexity theta(1)

function topElement():

if q.isEmpty()

throw Exception "Queue is empty" Complexity theta(1)

topElement <- q.elems[q.front]

end

function isFull()

if q.firstEmpty = -1

isFull <- true Complexity theta(1)

isFull <- false

function isEmpty():

if q.front = -1

isEmpty <- true Complexity theta(1)

isEmpty <- false

***Solution implementation***

N -size of the matrix

Maze :

Stack s

Queue q

Icteger a[N][N]

function Maze (m,a[N][N]):

m.s.init()

m.q.init()

for i <- 0,N

for j <- 0,N

m.a[i][j] <- a[i][j]

end

Complexity O(n\*n)

function isValid(mat[][N], visited[][N], row, col):

if row >= 0 and row < N and col >= 0 and col < N and mat[row][col]

and !visited[row][col]

isValid <- true

isValid <- false

end

Complexity theta(1)

function BFS(m,mat[][N, i, j, x, y, solutie[N][N]):

m.s.init()

m.q.init();

visited[i][j] = true

nod{ i,j }

m.q.push(nod)

while q.isEmpty() = false

node <- m.q.pop()

i <- node.get\_line()

j <- node.get\_column()

if i = x and j == y

break

StackNode nd{ -1,0 }

StackNode nd2{ 0,-1 }

StackNode nd3{ 0,1 }

StackNode nd4{ 1,0 }

m.s.push(nd4)

m.s.push(nd3)

m.s.push(nd2)

m.s.push(nd)

for k <- 0, 4

x <- m.s.pop();

if isValid(mat, visited, i + x.get\_line(), j + x.get\_column())

visited[i + x.get\_line()][j + x.get\_column()] = true

QueueNode r{ i + x.get\_line(), j + x.get\_column() }

m.q.push(r)

end for

end while

for i <- 0, N

for j <- 0, N

if visited[i][j] = true

solutie[i][j] <-1

else

solutie[i][j] <- 0

end for

end

Complexity O(n\*n)

function printSolution(sol[N][N]):

for i <- 0, N

for j <- 0, N

print sol[i][j]

end

Complexity O(n\*n)

function isSafe(maze[N][N x, y):

if x >= 0 and x < N and y >= 0 and y < N and maze[x][y] = 1

isSafe <-true

isSafe <- false

end

Complexity theta(1)

function solveMazeUtil(maze[N][N], x, y, sol[N][N], sx, sy)

if x = sx and y = sy

sol[x][y] <- 1

solveMazeUtil <- true

if isSafe(maze, x, y) = true

sol[x][y] <- 1

if q.isEmpty() = false

node = q.pop()

if solveMazeUtil(maze, node.get\_line(), node.get\_column(),

sol,sx, sy) == true

solveMazeUtil <- true

sol[x][y] <- 0

solveMazeUtil <- false

return false

end

Complexity theta(1)

function solveMaze(maze[N][N], sx, sy)

for i <- 0, N

for j <- 0, N

if maze[i][j] = 1

nod{ i,j };

q.push(nod);

end for

end for

if solveMazeUtil(maze, 0, 0, sol, sx, sy) = false

print Solution doesn't exist\n"

solveMaze <- false

print "The walk through the maze : \n\n"

printSolution(sol)

solveMaze <- true

end

Complexity O(n\*n)

function solveMazeUtilShort(maze[N][N], x, y, sol[N][N], sx, sy)

if x = sx and y = sy

sol[x][y] <- 1

solveMazeUtil <- true

if isSafe(maze, x, y) = true

sol[x][y] <- 1

if q.isEmpty() = false

node = q.pop()

if solveMazeUtilShort(maze, x+1, y,sol,sx, sy) == true

solveMazeUtilShort <- true

if solveMazeUtilShort(maze, x, y+1,sol,sx, sy) == true

solveMazeUtilShort <- true

sol[x][y] <- 0

solveMazeUtil <- false

return false

end

Complexity theta(1)

function solveMazeShort(maze[N][N], sx, sy)

for i <- 0, N

for j <- 0,N

sol[i][j] <- 0

end for

end for

for i <- 0, N

for j <- 0, N

if maze[i][j] = 1

nod{ i,j }

q.push(nod)

end for

end for

if solveMazeUtilShort(maze, 0, 0, sol, sx, sy) = false

print Solution doesn't exist\n"

solveMaze <- false

print "The walk through the maze : \n\n"

printSolution(sol)

solveMaze <- true

end

Complexity O(n\*n)

void test()

{

Stack s;

s.init();

assert(s.isEmpty());

StackNode x;

//s.get\_firstEmpty();

// s.get\_top();

x= s.topElement();

try { x = s.pop(); }

catch(Exception){}

int cap= s.get\_cap();

for (int i = 0; i <= cap; i++)

{

try {

StackNode nod{ i,i };

s.push(nod);

}

catch (Exception) {}

}

for (int i = 0; i <= cap\*3; i++)

{

try { x = s.pop(); }

catch (Exception) {}

}

assert(s.isFull()==false);

s.get\_top();

s.isEmpty();

Queue q;

q.init();

assert(q.isEmpty());

try { q.pop(); }

catch(Exception){}

try { QueueNode qn = q.topElement(); }

catch (Exception) {}

QueueNode qn2{ 1,1 };

q.push(qn2);

//QueueNode qn1 = q.pop();

QueueNode qn3 = q.topElement();

}