Path in a maze

ADT Queue:

Domain:
Q = {q | q is a queue with elements of type TElem}

Interface:

- init(q)
  - **Description**: creates a new empty queue
  - **Pre**: True
  - **Post**: $q \in Q$, q is an empty queue

- destroy(q)
  - **Description**: destroys a queue
  - **Pre**: $q \in Q$
  - **Post**: q was destroyed

- push(q, e)
  - **Description**: pushes (adds) a new element to the rear of the queue
  - **Pre**: $q \in Q$, e is a TElem
  - **Post**: $q' \in Q$, $q' = q \oplus e$, e is the element at the rear of the queue

- pop(q)
  - **Description**: pops (removes) the element from the front of the queue
  - **Pre**: $q \in Q$
  - **Post**: *pop* ← e, e is a TElem, e is the element at the front of q, $q' \in Q$,$q' = q \setminus e$
  - **Throws**: an underflow error if the queue is empty

- front(q)
  - **Description**: returns the element from the front of the queue(but it does not change the queue)
  - **Pre**: $q \in Q$
  - **Post**: *front* ← e, e is a TElem, e is the element from the front of q
  - **Throws**: an underflow error if the queue is empty

- isEmpty(q)
  - **Description**: checks if the queue is empty (has no elements)
  - **Pre**: $q \in Q$
  - **Post**: $isEmpty \leftarrow \begin{cases} \textit{true}, \text{ if q has no elements} \\ \textit{false}, \text{ otherwise} \end{cases}$

ADT Priority Queue:

Domain:
PQ = {pq | pq is a priority queue with elements(e, p), e $\in$TElem, p $\in$TPriority}

Interface:

- init(pq, R)
  - **Description**: creates a new empty priority queue
  - **Pre**: R is a relation over priorities,
    R : TPriority × TPriority
  - **Post**: pq $\in$PQ, pq is an empty priority queue

- destroy(pq)
  - **Description**: destroys a priority queue
  - **Pre**: pq $\in$PQ
  - **Post**: pq was destroyed

- push(pq, e, p)
  - **Description**: pushes (adds) a new element to the priority queue
  - **Pre**: pq $\in$PQ, e $\in$TElem, p $\in$TPriority
  - **Post**: pq′$\in$PQ, pq′ = pq $\oplus$ (e, p)

- pop(pq, e, p)
  - **Description**: pops (removes) from the priority queue the element with the highest priority. It returns both the element and its priority
  - **Pre**: pq $\in$PQ
  - **Post**: e $\in$TElem, p $\in$TPriority, e is the element with the highest priority from pq, p is its priority. pq′ $\in$PQ, pq′ = pq \ (e, p)
  - **Throws**: an underflow error if the queue is empty

- top(pq, e, p)
  - **Description**: returns from the priority queue the element with the highest priority and its priority. It does not modify the priority queue
  - **Pre**: pq $\in$PQ
  - **Post**: e $\in$TElem , p $\in$TPriority , e is the element with the highest priority from pq,p is its priority
  - **Throws**: an underflow error if the queue is empty

- isEmpty(pq)
  - **Description**: checks if the priority queue is empty (has no elements)
  - **Pre**: pq $\in$PQ
  - **Post**: *isEmpty* $\leftarrow \begin{cases} \textit{true}, \text{ if pq has no elements} \\ \textit{false}, \text{ otherwise} \end{cases}$

ADT Queue:

Representation:

QNode:
next: Integer
prev: Integer
info: TElem

Queue:
elems: QNode[]
cap: Integer
head: Integer
tail: Integer
firstEmpty: Integer

ADT Priority Queue:

Representation:

PQNode:
next: ↑PQNode
prev: ↑PQNode
info: TElem
priority: TPiority

PriorityQueue:
head: ↑PQNode
tail: ↑PQNode
r: Relation over priorities defined on TPriority × TPriority

Problem Statement:

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/or Queue (implemented on a doubly linked list on an array) and/or Priority Queue (implemented on a doubly linked list with dynamic allocation) - implement and use at least two ADTs .

The given ADTs are suitable for this problem because they are a good method of keeping track of the next positions to be visited in the path finding algorithm. We use this ADTs instead of others because they give a specific order to the data, which suits the path finding algorithm. According to which ADT we choose, the algorithm behaves differently.

The path finding algorithm starts from the initial position, checks all the accessible neighbours and adds them to the chosen ADT. According to the ADT one of them is chosen to be the next node. Again we check the neighbours and add them to the queue ot prioirty queue and again we select a new next position. The algorithm continues until there are no accessible positions left, or we managed to get out of the maze.

Implementation of the ADTs:

Queue:

```
subalgorithm init(q) is:
	q ← @create new Queue
	q.cap ← @initial size
	q.head ←-1
	q.tail ←-1
	q.firstEmpty ←0
	q.elems ←@create an array of QNodes of q.cap elements
	for i←0, q.cap - 1 execute
		q.elems[i].next ← i + 1
		q.elems[i].prev ← i – 1
	end-for
	q.elems[q.cap – 1].next = -1
end-subalgorithm
```

Complexity : Θ(n) – n being the capacity

```
sugalgorithm destroy(q) is:
	@delete the array of QNodes
end-subalgorithm
```

Complexity : Θ(1)

```
subalgorithm resize(q) is:
	newElems ←@create a new array of QNodes of 2 * q.cap elements
	for i ←q.cap -1 execute
		newElems[i] ←q.elems[i]
	end-for
	q.cap ← 2 * q.cap
	for i ←q.cap / 2, q.cap - 1 execute
		newElems[i].next ← i + 1
		newElems[i].prev ← i – 1
	end-for
	newElems[q.cap – 1].next ←-1
	@delete q.elems
	q.elems ←newElems
end-subalgorithm
```

Complexity : Θ(n) - n being the capacity

```
subalgorithm push(q, e) is:
        if q.firstEmpty = -1 then
                resize(q)
        end-if
        aux ← new Integer
        aux ← q.firstEmpty
        q.firstEmpty ← q.elems[q.firstEmpty].next
        q.elems[aux].info ← e
        if q.head = -1 then
                q.head ← aux
                q.tail ← aux
                q.elems[aux].next ← -1
                q.elems[aux].prev ← -1
        else
                q.elems[aux].next ← -1
                q.elems[tail].next ← aux
                q.elems[aux].prev ← tail
                q.tail ← aux
        end-if
end-subalgorithm
```

Complexity : Θ(1) (amortised, because when there is the need for resize the operation actually takes O(n), with n being the capacity)

```
function pop(q) is:
        if q.head = -1 then
                @throw underflow exception
        end-if
        if q.head = q.tail then
                q.tail ← -1
        end-if
        newHead ← new Integer
        newHead ← q.elems[q.head].next
        q.elems[q.head].next ← q.firstEmpty
        q.elems[q.head].prev ← -1
        q.firstEmpty ← q.head
        q.head ← newHead
        pop ← q.elems[q.firstEmpty].info
end-function
```

Complexity : Θ(1)

```
function front(q) is:
        if q.head = -1 then
                @throw underflow exception
        end-if
        front ← q.elems[q.head].info
end-function
```

Complexity : Θ(1)

```
function isEmpty(q) is:
	if q.head = -1 then
		isEmpty ← true
	isEmpty ← false
end-function
```

Complexity : Θ(1)

PriorityQueue:

```
subalgorithm init(pq, R) is :
	pq ← @create new PriorityQueue
	pq.head ← NIL
	pq.tail ← NIL
	pq.r ← R
end-subalgorithm
```

Complexity : Θ(1)

```
subalgprithm destroy(pq) is:
	while pq.head ≠ NIL execute
		pq.tail ← pq.head.next
		@delete pq.head
		pq.head ← pq.tail
	end-while
end-subalgorithm
```

Complexity : Θ(n) – n being the number of elements in the queue

```
subalgorithm push(pq, e, p) is:
	newNode ← allocate()
	[newNode].info ← e
	[newNode].priority ← p
	[newNode].next ← NIL

	if pq.head = NIL then
		[newNode].prev ← NIL
		pq.head ← newNode
		pq.tail ← pq.head
		@exit subalgorithm
	end-if
	aux ← pq.head
	while aux ≠ NIL and pq.r([aux].priority, p) execute
		aux ← [aux].next
	end-while
	if aux = NIL then
		[newNode].prev ← pq.tail
```

```
            [pq.tail].next ←newNode
            pq.tail ←newNode
      else
            [newNode].prev ←[aux].prev
            [newNode].next ←aux
            if [aux].prev ≠ NIL then
                  [[aux].prev].next ←newNode
            else
                  pq.head ← newNode
            end-if
            [aux].prev ←newNode

end-subalgorithm
```

Complexity : O(n)

```
subalgorithm pop(pq, e, p) is:
      if pq.head = NIL then
            @throw underflow exception
      end-if
      e ←[pq.head].info
      p ←[pq.head].priority
      aux ← pq.head
      pq.head ← [pq.head].next
      if pq.head = NIL then
            pq.tail ←pq.head
      else
            [pq.head].prev ←NIL
      end-if
      @delete aux

end-subalgorithm
```

Complexity : Θ(1)

```
subalgorithm top(pq, e, p) is:
      if pq.head = NIL then
            @throw underflow exception
      end-if
      e ←[pq.head].info
      p ←[pq.head].priority

end-subalgorithm
```

Complexity : Θ(1)

```
function isEmpty(pq) is:
      if pq.head = NIL then
            isEmpty ←true
      isEmpty←false
```

end-function
Complexity : Θ(1)

In order to solve the problem we will use two algorithms for path finding similar to the Breath-first search and A* from graphs. To keep the positions in the maze in a single variable we will define a new structure.

<u>Point:</u>
x: Integer
y: Integer

<u>Interface for the solving functions:</u>

- **readInput**(M, initX, initY, n, m):
  - **Description**: reads the input from the file and sets up a new matrix with the coresponding input data, the empty positions will be 0, the others will be -1
  - **Pre**: true
  - **Post**: M is a new 2 dimensional array of n * m dimension, n is the number of rows in the matrix, m is the number of columns, initX is the initial x position of the robot, initY is the initial y position of the robot

- **bfs**(M, initX, initY, exitX, exitY, n, m):
  - **Description**: searches the matrix M for an exit using an algorithm similar to a breath-first search
  - **Pre**: M is a 2 dimensional array, n and m are the dimensions of the matrix, initX and initY are the initial positions of the robot
  - **Post**: the matrix is changed such that each empty(0) position now has the number of steps necessary to reach that position from the initial position

- **aStar**(M, initX, initY, exitX, exitY, n, m):
  - **Description**: searches the matrix M for an exit using an algorithm similar to A*, it picks with a bigger priority the positions closer to edges of the matrix
  - **Pre**: M is a 2 dimensional array, n and m are the dimensions of the matrix, initX and initY are the initial positions of the robot
  - **Post**: the matrix is changed such that each empty(0) position now has the number of steps necessary to reach that position from the initial position

- **checkPath**():
  - **Description**: uses the readInput and aStar functions to decide whether or not the is a way out of the matrix
  - **Pre**: true
  - **Post**: $checkPath \leftarrow \begin{cases} \textit{true}, \text{ if there is a way out the input matrix} \\ \textit{false}, \text{ otherwise} \end{cases}$

- **retrivePath**(M, initX, initY, endX, endY, n, m):
  - **Description**: uses the modified(after the bfs or a*) matrix to determine the path the robot had to take to get out of the maze
  - **Pre**: M is the modified matrix, initX and initY are the initial positions of the robot, endX and endY are the positions of the robot immediately after exiting the maze, n and m are the dimensions of the maze
  - **Post**: the path will be printed on the screen

- **calculateHeuristic**(x, y, n, m):
  - **Description**: calculates the priority of a position according to how close it is to the edges of the maze
  - **Pre**: x and y is the current position, n and m the dimensions of the matrix
  - **Post**: calculateHeuristic ← the priority of the position (x, y)
- **path**():
  - **Description**: calculates a way out the maze using aStar, and prints the found path
  - **Pre**: true
  - **Post**: the path will be printed on the screen, or if there is no path a message will be printed

- **lowestPath**():
  - **Description**: calculates a way out the maze using bfs, and prints the found path, this algorithm assures us of a lowest length path because it explores the entire matrix
  - **Pre**: true
  - **Post**: the path will be printed on the screen, or if there is no path a message will be printed

<u>Implementation of the solving functions:</u>

We will need in order to solve the problem two special vectors which let us calculate the neighbours of a position easier.
lX ← [-1, 0, 1, 0], lY ← [0, -1, 0, 1]

subalgorithm **readInput**(M, initX, initY, n, m) is:

```
    f ← @open the input file
    n ← @read from f
    m ← @read from f
    M ← @a new 2 dimensional array of (n + 2) * (m + 2) dimensions, with 0 on every position

    for i ← 1, n execute
        for j ← 1, m execute
            c ← @read from f
            if c = 'X' then
                M[i][j] ← -1
            end-if
```

```
                if c = 'R' then
                        initX ← i
                        initY ← j
                end-if
        end-for
    end-for
end-subalgorithm
Complexity:  Θ(n * m)

sublagorithm bfs(M, initX, initY, exitX, exitY, n, m) is:
    Q ← @new Queue of points
    init(Q)
    newX ← @Integer
    newY ← @Integer
    initialP ← @Point
    initialP.x ← initX
    inittialP.y ← initY
    M[initX][initY] ← 1
    Q.push(initialP)

    while not Q.isEmpty() execute
        aux ← Q.pop()
        for i ← 0, 3 execute
            newX ← aux.x + lX[i]
            newY ← aux.y + lY[i]
            if M[newX][newY] = 0 then
                //available position
                M[newX][newY] ← M[aux.x][aux.y] + 1
                if newX = 0 or newX > n or newY = 0 or newY > m then
                    //we managed to exit the maze
                    exitX ← newX
                    exitY ← newY
                    @exit subalgorithm
                end-if
                initialP.x ← newX
                initialP.y ← newY
                Q.push(initialP)
            end-if
        end-for
    end-while
    exitX ← -1
    exitY ← -1
end-suablgorithm
Complexity:  Θ(n * m)

function calculateHeuristic(x, y, n, m) is:
    val ← y
```

```
        val ← min(val, m – y)
        val ← min(val, n – x)
        val ← min(val, x)
        calculateHeuristic ← val
end-function
```

Complexity: Θ(1)

sublagorithm **aStar**(M, initX, initY, exitX, exitY, n, m) is:

```
        PQ ← @new priority queue of points with priorities Integer
        init(PQ, <) //the relation between the priorities is “less”
        newX ← @Integer
        newY ← @Integer
        cost ← @Integer
        initialP ← @Point
        initialP.x ← initX
        inittialP.y ← initY
        M[initX][initY] ← 1
        PQ.push(initialP, calculateHeuristic(initX, intiY, n, m))

        while not PQ.isEmpty() execute
                aux ← @Point
                PQ.pop(aux, cost)
                for i ← 0, 3 execute
                        newX ← aux.x + lX[i]
                        newY ← aux.y + lY[i]
                        if M[newX][newY] = 0 then
                                //available position
                                M[newX][newY] ← M[aux.x][aux.y] + 1
                                if newX = 0 or newX > n or newY = 0 or newY > m then
                                        //we managed to exit the maze
                                        exitX ← newX
                                        exitY ← newY
                                        @exit subalgorithm
                                end-if
                                initialP.x ← newX
                                initialP.y ← newY
                                PQ.push(initialP, calculateHeuristic(newX, newY, n, m))
                        end-if
                end-for
        end-while
        exitX ← -1
        exitY ← -1
end-suablgorithm
```

Complexity: $O(n^2 * m^2)$ – the complexity is so high because it takes n * m for the algorithm to traverse the matrix and the extra m * n complexity comes from the implementation of the PriorityQueue, but the algorithm behaves much better than that in practice, because it searches in a greedy way for a path out of the maze.

```
function checkPath() is:
        n ←@Integer
        m ←@Integer
        initX ←@Integer
        initY ←@Integer
        endX ←@Integer
        exitY ←@Integer
        M ←@a two-dimensional array

        readInput(M, initX, initY, n, m)
        aStar(M, initX, initY, exitX, exitY, n, m)
        @delete M
        if exitX = -1:
                checkPath ←false
        checkPath ←true
end-function
```
Complexity: $O(n^2 * m^2)$

```
void retrivePath(M, initX, initY, endX, endY, n, m) is:
        if initX = endX and initY = exitY then
                @print position (initX, initY)
                @exit subalgorithm
        end-if
        for i ←0, 3 execute
                newX ←exitX + lX[i]
                newY ←exitY + lY[i]

                if newX > 0 and newX <= n and newY > 0 and newY <=m and
                   M[newX][newY] ← M[exitX][exitY] – 1 then
                        retrivePath(M, initX, initY, newX, newY, n, m)
                        @print the position (exitX, exitY)
end-subalgorithm
```
Complexity: O(n + m) – the algorithm makes (4 * the length of the path) stepts to find the path, and because the maximum length in a maze is n + m we can consider this as a valid complexity.

```
subalgorithm path() is:
        n ←@Integer
        m ←@Integer
        initX ←@Integer
        initY ←@Integer
        endX ←@Integer
        exitY ←@Integer
        M ←@a two-dimensional array

        if not checkPath() then
```

```
        @print a no path message
    end-if

    readInput(M, initX, initY, n, m)
    aStar(M, initX, initY, exitX, exitY, n, m)
    retrivePath(M, initX, initY, endX, endY, n, m)

    @delete M
end-subalgorithm
```
Complexity: $O(n^2 * m^2)$

subalgorithm **lowestPath**() is:

```
    n ←@Integer
    m ←@Integer
    initX ←@Integer
    initY ←@Integer
    endX ←@Integer
    exitY ←@Integer
    M ←@a two-dimensional array

    if not checkPath() then
        @print a no path message
    end-if

    readInput(M, initX, initY, n, m)
    bfs(M, initX, initY, exitX, exitY, n, m)
    retrivePath(M, initX, initY, endX, endY, n, m)

    @delete M
end-subalgorithm
```
Complexity: $\Theta(n * m)$

<u>Complexity of the push operation of PriorityQueue:</u>

Best case: The new element has a higher priority than the previous root. If that happens we just have to set the element to be the new root.
=> Θ (1)

Worst case: The new element has the lowest priority from the queue. So we have to set it to be the last element. In order to do that we have to go over all the elements.
=> Θ (n) – n being the number of elements in the priority queue

Average case:

$$\frac{(1+2+3+...+n+n)}{n+1}=\frac{\sum(i),(i=1,n)+n}{n+1}=\frac{\frac{n(n+1)}{2}+n}{n+1}=\frac{(n+1)}{n+1}*\frac{n}{2}*\frac{n}{n+1}=\frac{n^2}{2*n+2}\epsilon O(n)$$

Tests for the ADTs:

```
void smallTestQueue() {
	Queue<int> q;
	assert(q.isEmpty());
	for (int i = 0; i < 15; ++i) {
		q.push(i);
	}
	assert(!q.isEmpty());
	for (int i = 0; i < 15; ++i) {
		assert(q.front() == i);
		assert(q.pop() == i);
	}
	for (int i = 0; i < 5; ++i) {
		q.push(i);
	}
	for (int i = 0; i < 5; ++i) {
		assert(q.front() == i);
		assert(q.pop() == i);
	}
	try {
		q.pop();
		assert(false);
	} catch (...) {
		assert(true);
	}
	try {
		q.front();
		assert(false);
	} catch (...) {
		assert(true);
	}
}

void smallTestPQ() {
	PriorityQueue<int, int> PQ;
	assert(PQ.isEmpty());
	for (int i = 10; i >= 0; --i) {
		PQ.push(i, i);
	}
	assert(!PQ.isEmpty());
	int a, b;
```

```
	for (int i = 0; i <= 10; ++i) {
		PQ.top(a, b);
		assert(a == i && b == i);
		PQ.pop(a, b);
		assert(a == i && b == i);
	}
	try {
		PQ.top(a, b);
		assert(false);
	} catch (...) {
		assert(true);
	}
	try {
		PQ.pop(a, b);
		assert(false);
	} catch (...) {
		assert(true);
	}
	PQ.push(11, 11);
	PQ.push(12, 12);
	PQ.push(13, 13);
	PQ.push(15, 15);
	PQ.push(14, 14);
}
```