**MPA3 (Be a Maze, Be Very Amazed)**

**General Description**

Given a maze (composed of squares or cells that are either walls or open spaces) and the start (source) and end (destination) squares or cells, find a path solution from start (source) to end (destination).  If there is a solution, i.e. the destination is reachable from the source, display this solution, marking the cells along this path solution with an 'x'.  If there is no solution, display the original maze.

The input to this machine problem assignment is a file named **"maze.in"**. The first line of the input is a number representing the number of test cases or mazes to solve followed by the actual test cases.  Each test case is composed of the dimension of the maze and the maze itself.  The dimension of the maze is written in one line containing two integers delimited (separated) by a space. The first number represents the number of rows and the second, the number of the columns. This is followed by the actual maze in the next line. Below is a sample input file.

1  
4 7  
#######  
#...#o#  
#\*#...#  
#######

Each square or cell in the maze is either a wall, an open space, the source cell or the destination cell.  The '#' represents a wall, a cell that is not reachable or "walkable".  A period ( . ) represents an open space, a cell that is reachable or "walkable".  The start (source) cell is represented by a lower case 'o'.  And finally, the destination cell is represented by an asterisk (\*).

The goal is to find a path that will outline a solution to the maze problem.  This entails moving from one cell to another, starting from the source cell. Moving from one cell to another can only be done when they are adjacent. A cell at row***i*** and column ***j*** is adjacent to all cells found at (i, j-1) - left, (i-1,j-1) - up, (i, j+1) - right and (i+1,j) - down. These are the cells directly to its left, above it, to its right and below it. The source and destination cells are always located in an open space cell. A move can only be done to an adjacent open space cell. There may be more than one possible move from a cell in (i,j). Always choose the cell adjacent to its left, then the cell above, followed by the cell to its right and finally the cell below it.

Once a path is found, the cells along this path should be marked with a lower case 'x'. Below is a sample output. There should be two outputs for every test case, a solution found via a [queue](http://vle.upcebu.edu.ph/mod/folder/view.php?id=249) and another via a [stack](http://vle.upcebu.edu.ph/mod/folder/view.php?id=252).

#######  
#xxx#o#  
#\*#xxx#  
#######  
  
#######  
#xxx#o#  
#\*#xxx#  
#######

**Implementation Details**

**cell or square**

square (or cell) should represent a square in a maze. It should at least have the following members. You may have all of them specified as public. What are the needed information for the constructor?

* type of a square (maybe a character or an enumeration)
* location (row, column)
* visited (use this to make sure squares are visited at most once)
* constructor(parameter [list](http://vle.upcebu.edu.ph/mod/folder/view.php?id=203)?)

**maze**

maze should represent the maze. The maze has a size, its dimension (number of rows and columns).  The maze will simply be represented with a 2-dimensional array of squares, its size defined by its dimension. Notice that the size of the maze is variable. This means that the allocation of the 2d array should be dynamic. Also, figure out the parameter [list](http://vle.upcebu.edu.ph/mod/folder/view.php?id=203) of the constructor. It should have a tostring operation that returns the maze as a string.

* a 2d array of squares
* its dimension (how many rows and how many columns)
* constructor(parameter [list](http://vle.upcebu.edu.ph/mod/folder/view.php?id=203)?)
* tostring() - returns the maze as string

/\*for those who will be implementing this in c++  
c++ dynamic allocation of 2d array objects  
since the array must be dynamically allocated, the 2d array must be a pointer to a pointer of squares  
\*/  
squares \*\*themaze;  
  
/\*dynamic allocation\*/  
themaze = new squares\*[n] //this allocates memory for n rows  
//the next step is allocate the size of each row  
for(int i=0; i<n; i++)  
 themaze[i] = new squares[m]; //allocating m elements for each row  
  
//themaze is now ready to assigned with individual values  
  
  
/\*it is important to free this dynamically allocated memory to avoid memory leaks\*/  
  
//free each row first  
for(int i=0; i<n; i++)  
 delete[] themaze[i];  
//finally, free themaze  
delete[] themaze;

**agenda**

Two solutions must be provided for each test case, one using a [queue](http://vle.upcebu.edu.ph/mod/folder/view.php?id=249), and the other, using a [stack](http://vle.upcebu.edu.ph/mod/folder/view.php?id=252). There is a natural tendency to make two separate functions/methods to solve the maze, the first one with the use of a [queue](http://vle.upcebu.edu.ph/mod/folder/view.php?id=249) and the second with the use of a [stack](http://vle.upcebu.edu.ph/mod/folder/view.php?id=252). But a close inspection of these two separate solutions will reveal that they are doing exactly the same processes except for the data structure used.  To avoid doing two separate solutions, create a "wrapper" class that will have similar operations as the [stack](http://vle.upcebu.edu.ph/mod/folder/view.php?id=252) and the [queue](http://vle.upcebu.edu.ph/mod/folder/view.php?id=249). It will also hold a data member storing the type of adt used, i.e. either a [queue](http://vle.upcebu.edu.ph/mod/folder/view.php?id=249) or a [stack](http://vle.upcebu.edu.ph/mod/folder/view.php?id=252). You may include an int or an enumeration that determine its type ([stack](http://vle.upcebu.edu.ph/mod/folder/view.php?id=252) or [queue](http://vle.upcebu.edu.ph/mod/folder/view.php?id=249)). Call it agenda. And should provide the following operations:

* add - a facility to add an item depending on the adt used
* remove - a facility to remove an item depending on the adt used
* isempty - returns true if the adt is emply, false otherwise
* peek - returns the accessible item (top for the [stack](http://vle.upcebu.edu.ph/mod/folder/view.php?id=252), front for the [queue](http://vle.upcebu.edu.ph/mod/folder/view.php?id=249))
* getsize - returns the size of the adt

It should look like this in c++:

template <class T>  
class agenda{  
private:  
 T \*adt;  
 int type; //1 for [stack](http://vle.upcebu.edu.ph/mod/folder/view.php?id=252), 2 for [queue](http://vle.upcebu.edu.ph/mod/folder/view.php?id=249)  
public:  
 agenda();  
 ~agenda(){delete adt;}  
 void add(square);  
 bool remove();  
 bool isempty();  
 int getsize();  
 square peek();  
};  
  
template <class T>  
agenda<T>::agenda(){  
 adt = new T();  
 string t1 = typeid(\*adt).name();  
 string t2 = typeid([stack](http://vle.upcebu.edu.ph/mod/folder/view.php?id=252)<square>).name();  
 string t3 = typeid([queue](http://vle.upcebu.edu.ph/mod/folder/view.php?id=249)<square>).name();  
 //cout<<t1<<endl<<t2<<endl;  
 if(t1.compare(t2)==0)  
 type = 1;  
 else  
 if(t1.compare(t3)==0)  
 type = 2;  
}  
  
int main(){  
 agenda<[stack](http://vle.upcebu.edu.ph/mod/folder/view.php?id=252)<square> > a; //how to know what type of agenda is a?  
 agenda<[queue](http://vle.upcebu.edu.ph/mod/folder/view.php?id=249)<square> > b; //how to know what type of agenda is b?  
  
 return 0;  
}

typeid(<some\_object\_or\_class>).name() is a function that returns the type of an object/or a class as a string. You may use this know with what type the agenda is created. Refer to the constructor of agenda above.

**mazesolver**

mazesolver is the main class for solving a maze.  It will contain an agenda and the maze.  Figure out, as well, the parameter [list](http://vle.upcebu.edu.ph/mod/folder/view.php?id=203) of its constructor.  It will contain two public operations/methods namely solve and writesolution.  solve will find the path from source to destination, if there is any. writesolution will print the solution to a file named **"maze.out"**.  You may include other instance/member variables and operations/methods in mazesolver if the need arises.

Note: Whatever form of a solver it is ([stack](http://vle.upcebu.edu.ph/mod/folder/view.php?id=252) or [queue](http://vle.upcebu.edu.ph/mod/folder/view.php?id=249)), there must be a way for the solution to go back or backtrack when the solution reaches a dead end. This (backtracking) may be easily implemented using a [stack](http://vle.upcebu.edu.ph/mod/folder/view.php?id=252). This [stack](http://vle.upcebu.edu.ph/mod/folder/view.php?id=252) (different from the solver type [stack](http://vle.upcebu.edu.ph/mod/folder/view.php?id=252)) allows the solution to store previous moves with the most recent move always placed on top of the [stack](http://vle.upcebu.edu.ph/mod/folder/view.php?id=252).

class mazesolver{  
private:  
 agenda \*a;  
 maze \*m;  
public:  
 mazesolver();  
 ~mazesolver();  
 solve();  
 writesolution();  
};

**main or tester**

Below is exactly the order of testing or running the mazesolver.

int main(){  
 for each maze/test case  
 create instance of a maze  
 create instance of a mazesolver<[queue](http://vle.upcebu.edu.ph/mod/folder/view.php?id=249)> ms  
 ms.solve()  
 ms.writesolution()  
 create instance of a mazesolver<[stack](http://vle.upcebu.edu.ph/mod/folder/view.php?id=252)> ss  
 ss.solve()  
 ss.writesolution()  
}