COL106 - Data Structures and Algorithms
Assignment 2

Courtesy: Univ of Washington https://courses.cs.washington.edu/courses/cse373/13wi/homework/3/spec.pdf

Due: 26 August 2014, 11:55 pm

Total Marks: 30

This assignment focuses on implementation of a linear collection called a *deque* and an amusing problem that can be solved using it.
Please download the required files from the link given on the course webpage. You are required to complete the **TODO** sections in the `Main.cpp` for Section 1 and `DequeArray.cpp` for Section 2.

Note: A set of data files to use as input is also provided on the website. We do not guarantee that these form an exhaustive test. Perform additional testing of your own before you submit your program. A Makefile is also provided. Use `make` to compile all files. Use `make test` for compiling only the files required for Section 2. The default output will be `a.out`.

Submission Guidelines

Submit only 2 files: `Solve.txt` (It should have only the solve() definition) and `DequeArray.cpp` in a zipped format. Name your zipped file as ⟨EntryNumber⟩_⟨FirstName⟩_⟨LastName⟩.zip.

1 Maze Solver (10 Marks)

Input File: maze1.txt

```
 10 12
##########
#  S     #
# ### ## #
# #    # #
# #  # # #
# ## #####
# #      #
# # #  # #
##### ####
#   #    #
# #    #E#
##########
```

A deque (pronounced "deck") is an Abstract Data Type (ADT) for a double-ended queue. It is a linear ordered collection of elements that allows the user to add and remove elements from the front and back of the queue in constant ($O(1)$) time. Unlike a normal queue, which allows insertion only at the back and removal only from the front, a deque allows both at either end.
You will use a deque to implement an algorithm to search for a path to escape from a 2-D maze. We

are providing you with a `Maze` class representing the maze with various useful data and operations and `DequeADT` interface representing the Deque that you have to use. You will complete the **TODO** sections of `Main.cpp` so that it implements the algorithm described by the following pseudo-code:

Pseudocode for solve(Maze, DequeADT⟨ Point ⟩):

```
Keep a deque of squares to visit, initially empty.
Put the maze's start location in the deque.
Repeat the following until the deque is empty, or until we solve the maze:
  L := Remove the first location from the deque.
  If L is the maze's end location, then we have solved the maze.
  If we have already visited L before, or L is a wall, ignore it.
  Otherwise:
    Mark L as being visited.
    Add the neighbors of L (up 1, down 1, left 1, right 1) to your deque:
      If the neighbor is closer to the end location than L,
      add it to the front of the deque. Otherwise add it to the back.
If you end up with an empty deque and have not found the end, there is no solution.
```

The algorithm examines possible paths, looking for a path from the start(S) to the end(E). It is more efficient if it examines paths that move closer to the end first. So it takes advantage of the nature of a deque by adding closer neighbors to the front of the deque, meaning that they will be processed sooner, and further neighbors to the end of the deque, meaning that they will be processed later. Maze locations are returned to you as C++ `Point` objects that have a distance method you can use to see how far apart they are from each other. Note that we are using the standard screen coordinate system; "up" means negative y and "down" means positive y; "left" means negative x and "right" means positive x.

Implementation Details

The method you will implement in this section is named `solve()`. It is just one method; you must write it with exactly the header shown, so that it can be called by the provided `main()` within Main.cpp. You may define additional methods in your class if you like, but they must be private. (You probably should not need any data fields to implement this behavior.)

```
void solve(Maze * maze, DequeADT⟨ Point ⟩ * dq)
```

In this method, you will search for a path in the given maze from the start location to the end location using the deque-based algorithm described in the pseudocode, marking each square you visit along the way.

This method is called by the provided `main()`, passing you the maze for the input file the user indicates. If the maze passed is null, you should handle it in an appropriate manner (without changing any other part of the file). If the maze is non-null, you may assume that its state is completely valid, though some mazes do not have any successful path from the start to the end location.

The Maze class has the following methods that you can use:

- `public Point getStartPoint()`
  `public Point getEndPoint()`
  Returns the starting and ending locations in the maze, respectively. A `Point` object has private `row` and `column` fields representing its position as well as some useful methods such as distance.

- `public int getNumColumns()`
  `public int getNumRows()`
  Returns the dimensions of the maze. The previous example maze1.txt is 12 x 10 in size.

- `public boolean isInBounds (Point pt)`
  Returns true if the given point position is within the bounds of the maze. Use this to avoid out-of-bounds indexes. All of the other methods that accept point parameters check for bad indexes.

- `public void setVisited (Point pt)`
  Marks the given point position as having been visited by your algorithm.

- `public boolean isVisited(Point pt)`
  Returns true if the given point position has been previously marked as visited by a call to `setVisited`.

- `public boolean isWall(Point pt)`
  Returns true if the given point position stores a wall of the maze (a '#'character).

- `public void print()`
  Prints a text representation of the maze, which looks like its input file plus dots to mark any visited squares.

NOTE: *You can use the files maze1.txt, maze2.txt and maze3.txt and their respective solutions to test/verify your code.*

2 ArrayDeque (15 Marks)

In this part, you will write your own implementation of a deque. A deque can be implemented efficiently using either an array or a linked list of nodes; in our case you will use an array as the internal data structure. A key aspect of a well-implemented deque is that its common operations such as adding and removing elements from both ends must be fast; they must run in $O(1)$ average runtime. An unfilled array is efficient by nature for adding and removing at the end (highest indexes) but slow for removing from the front (index 0) because of the need to shift elements left by one to cover the hole left by the removed first element.
To get around this and implement the deque efficiently, your array will use a technique called a 'circular buffer'. This means that as elements are added and removed, not only does the size (or "last meaningful index") of the array change, but if they are added/removed from the front, the start of the data (or "first meaningful index") also changes.
For example, suppose we have a 10-element circular array buffer representing a deque of strings. It is initially empty and then we add six elements to it as shown by calling `push_back` and passing A, then B, C, D, E, and then F:

index	0	1	2	3	4	5	6	7	8	9
value	A	B	C	D	E	F				
size	6		*front*	0						

At this point the deque is [A, B, C, D, E, F]; a call to `front` would return A and `back` would return F. Now suppose the user calls `pop_front` three times, which returns A, then B, then C. The deque's state is now the following. Notice that D-F did not shift to index 0 but remain in their original locations:

index	0	1	2	3	4	5	6	7	8	9
value				D	E	F				
size	**3**		*front*	**3**						

At this point the deque is [D, E, F]; a call to `front` would return D and `back` would return F. To use all available array space, a circular buffer wraps around to its other end if needed. Suppose the client calls `pop_front` on D, then calls `push_back` six more times, adding G, H, I, J, K, and L. The deque's state is now:

index	0	1	2	3	4	5	6	7	8	9
value	**K**	**L**			E	F	**G**	**H**	**I**	**J**
size	**8**		*front*	**4**						

At this point the deque is [E, F, G, H, I, J, K, L]; a call to `front` would return E and `back` would return L. The wrapping occurs in both directions. Suppose we go back to our second diagram above that contained only D-F in indexes 3-5, and we called `push_front` five more times, adding M, N, O, P, and Q. The deque's state would be:

index	0	1	2	3	4	5	6	7	8	9
value	**O**	**N**	**M**	D	E	F			**Q**	**P**
size	**8**		*front*	**8**						

At this point the deque is [Q, P, O, N, M, D, E, F]; a call to `front` would return Q and `back` would return F. If the deque becomes full, enlarge it by copying its data into a new array twice as large. You cannot simply copy over the old contents into the same indexes in the new array because the wrapping changes. Instead, copy the deque from front to back starting at index 0 of the new array. If we start from the 8-element array just shown and use `push_back` to add R, S, and T, the T will be the 11th element and will therefore trigger a resize. After the resize the deque's internal state will be:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
value	**Q**	**P**	**O**	**N**	**M**	**D**	**E**	**F**	**R**	**S**	**T**									
size	**11**		*front*	**0**																

As one last example, if we then called `push_front` on this larger array and added U, V, and W, the deque would be:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
value	**Q**	**P**	**O**	**N**	**M**	**D**	**E**	**F**	**R**	**S**	**T**							**W**	**V**	**U**
size	**14**		*front*	**17**																

Implementation Details

You will write a class named `DequeArray⟨T⟩` that implements our provided `DequeADT⟨T⟩` interface. It has the following methods; you must write them with exactly the header shown, so it can be called by the provided classes. You may define additional methods in your class if you like, but they must be private. You will also need to declare any private data fields needed by the class in order to implement this behavior. All specified methods must run in $O(1)$ average runtime unless otherwise specified.

- `public DequeArray()`
  In this constructor, you will initialize a newly constructed empty deque. Give it a default array capacity of 10.

- `public void push_front(T& val)`
  `public void push_back(T& val)`
  In these methods, you should add the given element value `val` to the front or back of your deque respectively. If the element is null, you should handle it appropriately. (You may need to resize your array to fit the new element. Resizing is $O(N)$ but if you resize to a multiple of the old size, the average runtime for adding is $O(1)$.)

- `public void clear()`
  In this method, you should remove all elements from your deque. Make sure that your internal array stores only null values after a call to clear so that the memory previously occupied by the elements can be freed. This method is $O(N)$.

- `public boolean empty()`
  In this method, you should return whether your deque does not contain any elements.

- `public T front()`
  `public T back()`
  In these methods, you should return the first or last element of your deque respectively without modifying the state of the deque. If the deque does not contain any elements, you should handle it appropriately.

- `public void pop_front()`
  `public void pop_back()`

In these methods, you should remove and return the first or last element of your deque respectively. If the deque does not contain any elements, you should handle it appropriately. You should null out the now-empty array slot.

- `public int size()`
  In this method, you should return the number of elements in your deque.

- `public T at(int index)`
  This method should return the value at index specified by `index`.

NOTE: *You can use the file TestDeque.cpp to test/verify your code.*

3 Maze Solver Revisited (5 Marks)

Yes, you guessed it right! Now that you have your own `DequeArray` class that you have implemented in Section 2, re-run the Maze Solver described in Section 1 using it.