

**CS 201 DATA STRUCTURES**  
**ASSIGNMENT 3**  
**SECTION A&B**  
**Fall 2017**

**DUE:** Oct 3, 2017

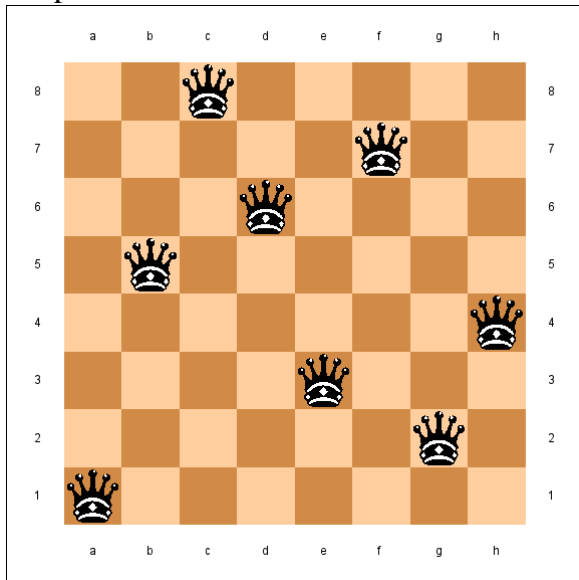
**NOTE:** Late submissions will not be accepted

**TO SUBMIT:** Documented and well written structured code in C++ on slate. Undocumented code will be assigned a zero.

**PROBLEM 1**

The **n queens puzzle** is the problem of placing  $n$  chess queens on an  $n \times n$  chessboard so that no two queens threaten each other. Thus, a solution requires that no two queens share the same row, column, or diagonal.

Below is one sample solution of 8 queens puzzle, more than one solutions may exists for this problem.



We want to compute all possible solutions of  $n$  queen puzzle. Unfortunately, no algorithm exists that can compute the solution in polynomial time. Consequently, we have to explore all possibilities and check whether the solution exists for our current choice or not.

Suppose queens are numbered from 1 to  $n$  and our solution can be represented by a one dimensional array *sol* such that  $sol[i]$  stores the column number of  $i^{th}$  queen in  $i^{th}$  row. For example the above solution of 8 queen problem can be represented as  $sol[1] = 3$ ,  $sol[2]=6$ ,  $sol[3] = 4$ ,  $sol[4] = 2$ ,  $sol[5] = 8$ ,  $sol[6] = 5$ ,  $sol[7] = 7$  and  $sol[8] = 1$ .


One possible way to explore all the possibilities is to place queens one by one in each row and check whether this placement is desired one or not. This means we will build partial solutions one at a time and proceed further if our partial solution is consistent with our constraints.

The board should be regarded as a set of constraints and the solution is simply satisfying all constraints. For example: queen1 attacks some positions, therefore queen 2 has to comply with these constraints and take place, not directly attacked by queen 1. Placing queen 3 is harder, since we have to satisfy constraints of queen1 and queen 2. Going the same way we may reach point, where the constraints make the placement of the next queen impossible. Therefore we need to relax the constraints and find new solution. To do this we are going backwards and finding new admissible solution. To keep everything in order we keep the simple rule: last placed, first displaced. In other words if we place successfully queen on the  $i^{\text{th}}$  column but cannot find solution for  $(i+1)^{\text{th}}$  queen, then going backwards we will try to find other admissible solution for the  $i^{\text{th}}$  queen first. Let's discuss this with example of 4 queen puzzle.



Algorithm:

- 1-Start with one queen at the first column first row
- 2-Continue with second queen from the first column second row
- 3-Increase column number until find a permissible situation and continue with next queen
- 4-If no permissible place is found change the position of previous queen
- 5- Out put the solution after placing the nth Queen and repeat step 3
- 6 - end the algorithm when no further position exists for queen 1



We place the first queen on column1 of row 1




now queen 2 can't be placed at column 1 and 2 of row 2. So we will place it at column 3 of row 2


Queen 3 can't be placed in any column of row 3 so increase the column (position) of Queen 2.



Now Queen 3 can't be placed in column 1 but column 2 is fine




Queen 4 can't be placed in any column of row 4 so we need to change the position of Queen 3. Also Queen 3 can't be placed in any next column (3 and 4). This means we have to increase the column of Queen 2 but Queen 2 is already at last column. Finally we will increase the position (column) of Queen 1.





Now Queen 2 can't be placed in column 1, 2 and 3 so we will place it in column 4

Next Queen 3 can be placed in column 1





			
			
			

Finally Queen 4 can be placed in column 3 of row 4

As we have placed the last queen so we have computed one solution of 4 Queen problem.

Another possible solution is

You can compute the solution of n queen problem using stack data structure. Whenever you place a queen i on the  $j^{\text{th}}$  column of  $i^{\text{th}}$  row of chess board, put the cell number (i,j) on stack and whenever you could not find any permissible location for the queen, pop the position of last queen placed on chess board, find the permissible location of previous queen and so on. Whenever you place the  $n^{\text{th}}$  queen on chess board, output the solution on screen. Do this until stack is not empty and this way you can compute all possible solutions of n queen puzzle.

Implement the class template stack with following functions as members

Push(T d) push data element d in the stack

Pop(T &d) Pop the element from stack and put it in d

Top(T &d) Get the top of the stack element and place it in d

IsEmpty() returns true if stack is empty and false otherwise

IsFull() returns true if stack is full and returns otherwise

Implement the function NQueen(int n) that outputs all possible solutions of n Queen puzzle using the stack.

## PROBLEM 2

Implement the class template Queue with the following functions

Enqueue(T d) insert the data item d at the end of the queue

Dequeue(T &d) remove the element at the front of the queue and place it in d

IsEmpty() returns true if queue is empty and false otherwise

IsFull() returns true if queue is full and returns otherwise

Implement the class Set that has following members

array of integers to store elements of the set

size of the set

function to output all the elements of the set  
function Add(int x) - add the element x at the end  
function Remove() removes the last element from the set  
function getLast() returns the last element of the set

Implement the function allSubset(int n) that outputs all the non-empty subsets of the set {1, 2, 3, ..., n} using the following algorithm that makes use of a queue.

- 1-Enqueue subset {1}
- 2-Dequeue a set *s* from the queue and output it
- 3-if *s*.getLast() < n then
  - 4-make a new set *r* same as set *s*
  - 5-remove last element of set *r*
  - 6-call *r*.Add(*r*.getLast()+1) and enqueue *r*
  - 7-call *s*.Add(*s*.getLast()+1) and enqueue *s*
- 8- repeat step 2 if queue is not empty

### **VERY IMPORTANT**

- Academic integrity is expected of all the students. Plagiarism or cheating in any assessment will result in negative marking or an **F** grade in the course, and possibly more severe penalties.