CS29003 Algorithms Laboratory

Assignment No: 5

Last date of submission: 12–February–2019

Stacks, Queues, and Trees

In this assignment, you solve two unrelated problems. The first one is on the realization of a queue, and the second one is on binary trees. In order to reduce your programming overhead, a blackbox BB5 is provided which provides the following utilities.

- A complete implementation of the stack ADT.
- A constructor of random binary trees.

The details of the blackbox features will be explained in appropriate places. In order use the blackbox, write following line after your usual **`#include`** directives.

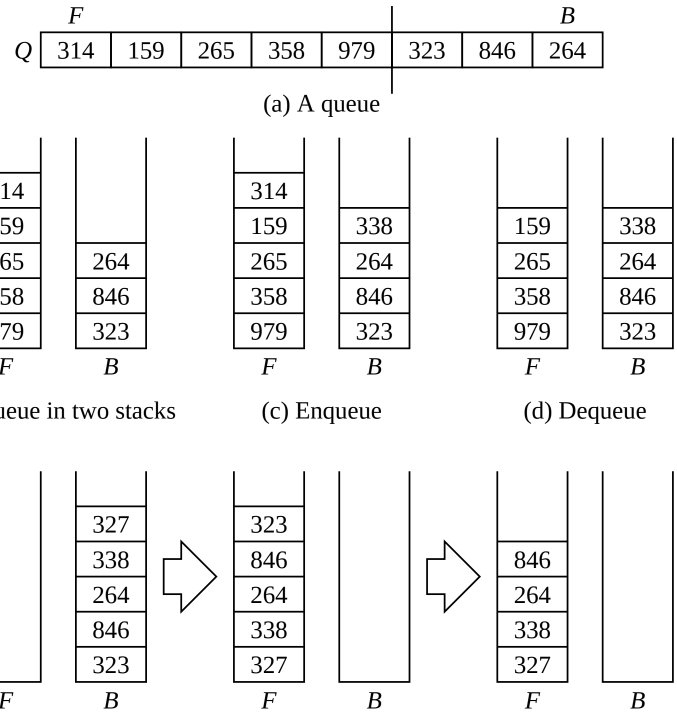
```
#include "BB5.h"
```

At the beginning of your **`main()`** function, call **`registerme()`** (this step is optional for this black box). Moreover, compile your code as:

```
gcc/g++ mycode.c/mycode.cpp BB5.o
```

Part 1: Implementation of a queue using two stacks

As the following figure illustrates, a queue Q (see Part (a) of the figure) can be realized from two stacks F and B (see Part (b)). An arbitrary break-point is chosen (between 979 and 323 in the figure). The part of Q before this break-point resides in the front stack F, and the part of Q after the break-point resides in the back stack B. Notice the order in which the elements of Q appear in F and B.



(a) A queue

(b) Queue in two stacks (c) Enqueue (d) Dequeue

(e) Dequeue from an empty front stack

An enqueue operation involves pushing the new item to the back stack B (see Part (c)). A dequeue operation is the same as pop from the front stack F. If F was not empty before the pop, this is straightforward (see Part (d)). If both F and B are empty, then Q is empty too, and a dequeue from Q is not permitted. If F is empty but B contains one or more elements (see Part (e)), the element to dequeue lies at the bottom of B, and cannot be directly accessed. Make a sequence of pop operations from B and push operations of those elements to F, until B becomes empty. Now, a normal dequeue (pop from F) can be performed.

In order to implement a queue this way, use the implementation of the **STACK** ADT from BB5. You may now define your queue as:

```
typedef struct {
   STACK F, B;
} QUEUE;
```

For a stack **S**, the ADT calls supplied by BB5 are tabulated below. Only stacks of integers are supported.

`S = SINIT()`	Create an empty stack.
`ISEMPTY(S)`	Returns 1 or 0 depending on whether S is empty or not.
`TOP(S)`	Returns the element (an integer) at the top of S.
`S = PUSH(S,x)`	Push an integer x to the stack S.
`S = POP(S)`	Perform a pop operation from S.
`SPRNT2B(S)`	Print the elements of S from top to bottom.
`SPRNB2T(S)`	Print the elements of S from bottom to top.

You do not need to understand the implementation of the **STACK** ADT (since your teachers did it, the operations are efficient). Use the above calls to implement the queue ADT as follows.

`Q = QINIT()`	Create an empty queue.
`Q = ENQUEUE(Q,x)`	Enqueue an integer x to Q.
`Q = DEQUEUE(Q)`	Perform a dequeue operation on Q.
`QPRN(Q)`	Print the elements of Q from front to back.

In your `main()` function, read a small integer n from the user. Start with an empty queue Q. Make n enqueue and n dequeue operations on Q. Enqueue randomly generated integers to Q. Never make an attempt to dequeue from an empty queue. Print the queue after each operation.

Part 2: Level-by-level listing of keys in a binary tree

The blackbox BB5 defines a binary-tree data type as follows. This is declared in **BB5.h**.

```
typedef struct _treenode {
   int key;
   struct _treenode *L, *R, *N;
} treenode;

typedef treenode *TREE;
```

By calling

```
T = TGEN(n);
```

you can construct a random binary tree on n nodes with random key values. A pointer to the root node is returned by this call. In order to print the tree in a human-readable format, use the following call.

```
TPRN(T);
```

The tree constructor of BB5 keeps all the **N** pointers NULL. Write a function

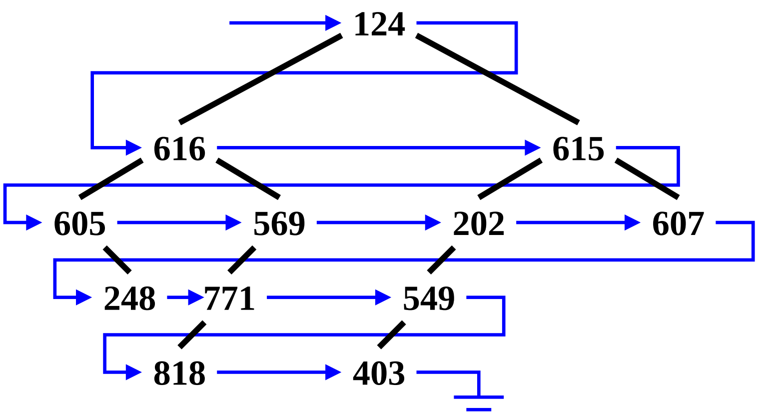
```
SETN(T)
```

to set these pointers as explained now. Let u be a node at level l in T. Impose a left-to-right ordering of the nodes in each level. If u is not the last node in T at level l, then the **N** pointer of u should point to the next node at the same level l. If u is the last node at level l, then the **N** pointer of u should point to the first node of the next level (or to NULL if l is the last level in T). Once the **N** pointers are so adjusted, T becomes a linked list with respect to these pointers starting at the root and storing the nodes in a level-by-level and left-to-right-in-each-level order. To implement this function, make a recursive traversal of T, and build a linked list of nodes at *each* level. After the traversal completes, join the level-wise lists into a single list. Write another function `TPRNL(T)` that makes a level-wise printing of the nodes of T. Use a linked-list print procedure following the **N** pointers.

In the `main()` function, read n (may be the same as in Part 1), build a tree on n nodes by **TGEN**, print the tree by **TPRN**, call **SETN** to set the **N** pointers, and then print the resulting linked list by calling **TPRNL**.

Sample output

```
n = 12

+++ Part 1

    QINIT()       : Q = [ ]
    ENQUEUE(525) : Q = [ 525 ]
    ENQUEUE(329) : Q = [ 525 329 ]
    ENQUEUE(590) : Q = [ 525 329 590 ]
    DEQUEUE()     : Q = [ 329 590 ]
    DEQUEUE()     : Q = [ 590 ]
    ENQUEUE(911) : Q = [ 590 911 ]
    DEQUEUE()     : Q = [ 911 ]
    ENQUEUE(297) : Q = [ 911 297 ]
    DEQUEUE()     : Q = [ 297 ]
    ENQUEUE(428) : Q = [ 297 428 ]
    ENQUEUE(388) : Q = [ 297 428 388 ]
    ENQUEUE(249) : Q = [ 297 428 388 249 ]
    DEQUEUE()     : Q = [ 428 388 249 ]
    DEQUEUE()     : Q = [ 388 249 ]
    ENQUEUE(431) : Q = [ 388 249 431 ]
    DEQUEUE()     : Q = [ 249 431 ]
    ENQUEUE(405) : Q = [ 249 431 405 ]
    ENQUEUE(336) : Q = [ 249 431 405 336 ]
    DEQUEUE()     : Q = [ 431 405 336 ]
    ENQUEUE(733) : Q = [ 431 405 336 733 ]
    DEQUEUE()     : Q = [ 405 336 733 ]
    DEQUEUE()     : Q = [ 336 733 ]
    DEQUEUE()     : Q = [ 733 ]
    DEQUEUE()     : Q = [ ]

+++ Part 2

--- Generated tree

    124
    L --> 616
    |      L --> 605
    |      |      L --> NULL
    |      |      R --> 248
    |      |             L --> NULL
    |      |             R --> NULL
    |      R --> 569
    |             L --> 771
    |             |      L --> 818
    |             |      |      L --> NULL
    |             |      |      R --> NULL
    |             |      R --> NULL
    |             R --> NULL
    R --> 615
           L --> 202
           |      L --> 549
           |      |      L --> 403
           |      |      |      L --> NULL
           |      |      |      R --> NULL
           |      |      R --> NULL
           |      R --> NULL
           R --> 607
                  L --> NULL
                  R --> NULL

--- Level-by-level printing

    124 616 615 605 569 202 607 248 771 549 818 403
```



Submit a single C/C++ source file. Do not use global/static variables.