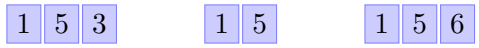
1. **[Pointers and Arrays] Stack and Queue [40 pts]**
   In this problem, you will implement two basic data structures using array: stack and queue. You can assume the elements are of type `int`.

   A stack is usually called "last in first out (LIFO) list". It supports two important operations, *push* and *pop*. Here's the list of all operations that you need to implement.

   - `void stack_init(stack *s, int capacity)`
     Initializes the stack with maximum size `capacity`.
   - `int stack_size(stack *s)`
     Returns the size of the stack, i.e. the number of elements in the array.
   - `int stack_pop(stack *s)`
     Returns the element on top of the stack. If the stack is empty, the return value is undefined.
   - `void stack_push(stack *s, int e)`
     If the stack is not full, push the item on top of the stack. Otherwise, do nothing.
   - `void stack_deallocate(stack *s)`
     Frees this stack.

   The `push` and `pop` operations of stack only operates on one end of the array. Figure 1 shows an example.

   (a) A stack (b) Pop (c) Push 6

   Figure 1: Illustration of Stack Operations

   A queue is usually called "first in first out (FIFO) list". It also supports two important operations, *enqueue* and *dequeue*. Here's the list of all operations that you need to implement.

   - `void queue_init(queue *q, int capacity)`
     Initializes the queue with maximum size `capacity`.
   - `int queue_size(queue *q)`
     Returns the size of the queue, i.e. the number of elements in the array.

- `int queue_dequeue(queue *q)`
  Returns the element at the front of the queue. If the queue is empty, the return value is undefined.
- `void queue_enqueue(queue *s, int e)`
  If the queue is not full, place the item at the back of the queue. Otherwise, do nothing.
- `void queue_deallocate(queue *s)`
  Frees this queue.

The `enqueue` operates at the end of the array and `dequeue` operations pops the first element of queue. Figure 2 shows an example.


(a) A queue (b) Dequeue (c) Enqueue 6

Figure 2: Illustration of Queue Operations

Create header file "`dslib.h`", which contains declarations of stack and queue, and implement them in "`stack.c`" and "`queue.c`". Test your program in "`test.c`", where the `main` function is. Also, create a "`Makefile`" to manage your project. Compress these five files into a zip file called "`code.zip`" and submit it to CMS [1].

2. **[Structs and Recursion] Binary Search Tree [40 pts]**
   In this problem, you will implement a very useful data structure called Binary Search Tree (BST). A Binary Search Tree is defined recursively as follows. (Assume each node has a unique key.)

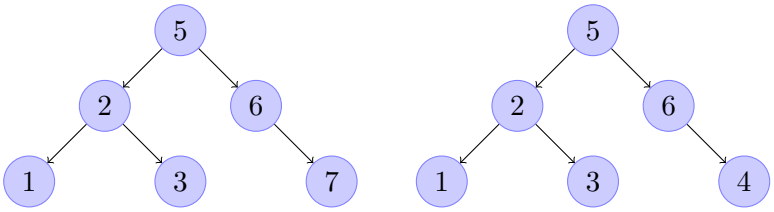
   - An empty tree is a BST.
   - The left subtree of a node contains only nodes with keys less than the node's key.
   - The right subtree of a node contains only nodes with keys greater than the node's key.
   - Both the left and right subtrees must also be BSTs.

   Figure 3 shows two examples.

   You will implement two functions for BST, `search` and `insert`.

   - `bst_node* search(bst *tree, int key)`
     Returns the pointer to the BST node, `NULL` if the key doesn't appear in the tree.

[1] `https://cms.csuglab.cornell.edu`




(a) A valid BST (b) An invalid BST

Figure 3: Illustration of BST

- `void insert(bst *tree, int key)`
  Inserts a node with the value `key` in the `tree`.

Create a header file call "`bst.h`" and implement Binary Search Tree in "`bst.c`". You are not allowed to use loop for `search` and `insert`, use recursion instead. Use "`Makefile`" to manage your project. Compress them into a zip file called "`bst.zip`" and submit it to CMS.

3. **[Union] Endianness of My Machine [20 pts]**
   Endianness is a difference in data representation at the hardware level. The most common cases refer to how bytes are ordered within a single 16-, 32-, or 64-bit word, and endianness is then the same as byte order. The usual contrast is whether the most-significant or least-significant byte is ordered first (at the smallest address) within the larger data item, which is known as **big-endian** and **little-endian** respectively. Figure 4 shows an example.


(a) Big-endian (b) Little-endian

Figure 4: How 0x90AB12CD is Stored on Different Machines

   In this problem, you are asked to write a program to test whether the host machine is big-endian or little-endian. Create a file called "`endianness.c`" and implement function "`int is_big_endian()`" in that file. The function returns non-zero value if the machine is big-endian and zero if it is little-endian. Submit "`endianness.c`" to CMS.

4. **[Bonus] Print 1 to 1000 [20 pts]**
   In this problem, you are asked to print number 1 to 1000, but you cannot use any loop or if-else statements. Furthermore, you cannot write 1000 printf statements. Write a `C` program under these restrictions. (You can email me for some hints.)