Asn 3 – Due July 22nd, 2016 (2:30pm)

Problem 1 [10 marks]

Consider the Queue ADT discussed in class along with its datum and list of operations:

- `print(Q)`
  Prints out the sequence of data from the front of the queue to the back.
- `enqueue(Q,x)`
  Puts the data x into the back of the queue
- `dequeue(Q)`
  Returns and removes the data from the front of the queue
- `isEmpty(Q)`
  Returns 0 (false) or 1 (true) depending on whether the queue is empty
- `size(Q)`
  Returns an unsigned integer representing the number of elements in the queue
- `peek(Q)`
  Returns the next data to be dequeued but doesn't actually dequeue it (it takes a 'peek')

You are to write a C program that creates a Queue datatype with an array of integers (among other attributes). Using the operations available for a C array, implement each one of the functions described above (some are more straight forward than others). Please note that C only has static arrays, your implementation must take that into account (especially when executing enqueue and dequeue).

Here is the start of your structure.

```
typedef struct {
      int * data;  // array of the data on the queue
      …
} queue_t;
```

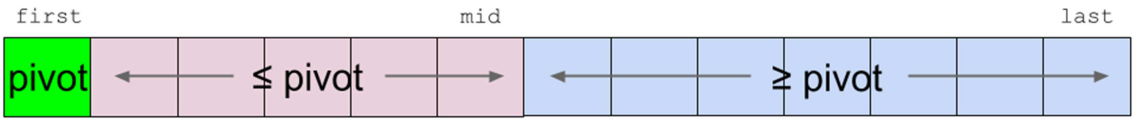
In your code, you should also include a test case that tests each one of these functions.

Problem 2 [14 marks]

```
int partition(int arr[], int first, int last) {
    int mid = first;
    int pivot = arr[first];
    for (int sweep = first+1; sweep <= last; sweep++) {
        // Assertion:  . . .
        if (arr[sweep] < pivot) {
            int tmp = arr[sweep];
            arr[sweep] = arr[mid+1];
            arr[mid+1] = tmp;
            mid++;
        }
    }
    // Post:  . . .
    arr[first] = arr[mid];
    arr[mid] = pivot;
    return mid;
}
```

The focus for this problem will be the main loop of partition. You will make arguments about what it attempts to accomplish.

a) [3 marks] The diagram depicts what we hope is the state of `arr[first..last]` just after the main loop terminates [but before the pivot is swapped into place]. Write down a post-condition that effectively summarizes the values and locations of the array elements, as they relate to indices `first`, `last` and `mid`.



b) [3 marks] Draw a similar diagram to depict the state of `arr[first..last]` at the *beginning* of each loop. Just like the provided diagram, place all of the indices in their correct positions, and indicate anything you might know about the values of the array elements themselves. If you don't know anything about some of the array elements, then indicate that too.

c) [3 marks] Write an assertion that is true at the beginning of each loop. Just like you did for part (a), it should summarize the values and locations of the array elements, as they relate to all indices, i.e., `first`, `last`, `mid`, but also **sweep**.

Hint: Your answer is probably correct if, when you substitute `sweep = last + 1`, it matches your answer to part (a).

d) [5 marks] Prove that the partition algorithm is correct using the initialization-maintenance-termination technique shown in class.

Problem 3 [6 marks]

Prefix Evaluation (unlike postfix described in class) has the evaluation strategy of [operation, data, data]. Whenever such a pattern is encountered, the operation is performed on the two data. For example

```
+  23  45
```

Indicates that we are performing the addition operation on the values 23 and 45 (result is 68). Write a pseudocode to obtain the result of a prefix expression using a queue. Your pseudocode should also determine when invalid prefix syntax has occurred. As in Ex. 3-12 in the notes, you may assume that the inputs for the prefix expression are given one by one (from beginning to end).

You may also assume the following operations are present in the "interface" of the queue data structure you are using:

- Enqueue(Q, data) - Puts the data x into the back of the queue
- Dequeue(Q) - Returns and removes the data from the front of the queue
- Size(Q) - Returns an unsigned integer representing the number of elements in the queue
- Peek(Q) - Returns the next data to be dequeued but doesn't actually dequeue it

Problem 4 [20 marks]

Complete the following three linked list functions according to the provided specifications. Use full C code. You may use the LL-node.h and LL.h files online as starting points ☺
In all cases, you should assume that a linked list data type is composed of two fields, a pointer to the head node, a pointer to the tail node, but both NULL if empty. Each node contains two fields: its value data and a pointer to the next node in the chain. Assume all linked lists and node sequences are valid.

In addition, you should write a main function that creates test cases for each function. Choose your test cases wisely such that you will have "full test coverage". For example, if you have code that looks like the following:

```
if (condition1){
      // Block1
} else if (condition2) {
      // Block2
} else {
      // Block3
}
```

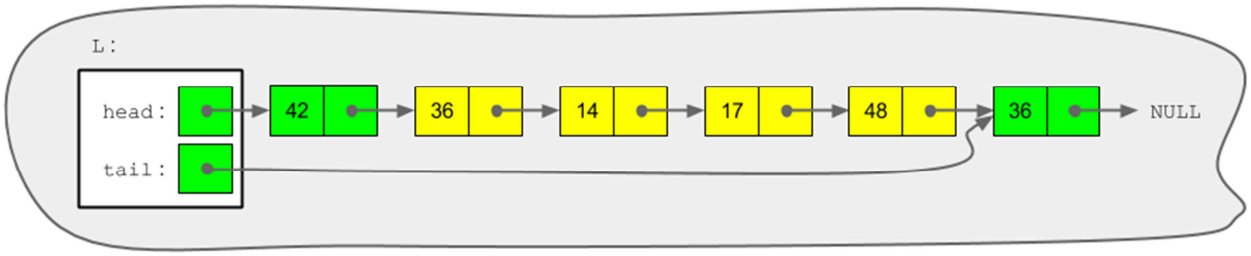
then you should write three test cases that will run through **all three Blocks of code** (`Block1`, `Block2` and `Block3`). Your test cases can simply use the print function to verify the result. A sample test for the `spliceinto()` function described below can look something like:

```
>>
L = 42 36 14 17 48 36
spliceinto(L, 53, 4)
L = 42 36 14 17 53 48 36
>>
```

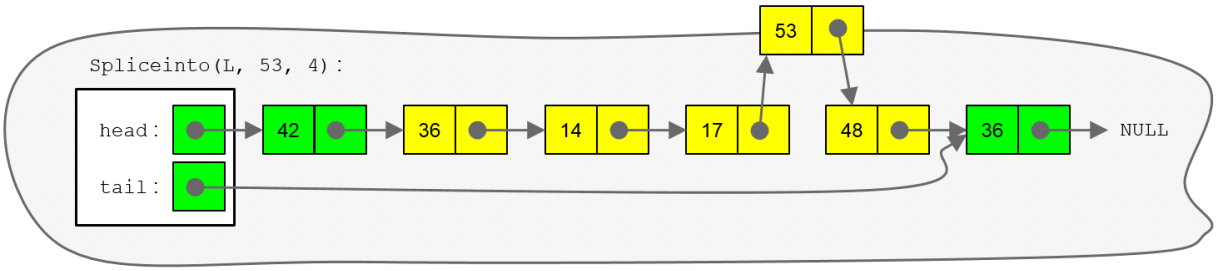
Asn 3 – Due July 22nd, 2016 (2:30pm)

Sample

Suppose we have the following input list L:

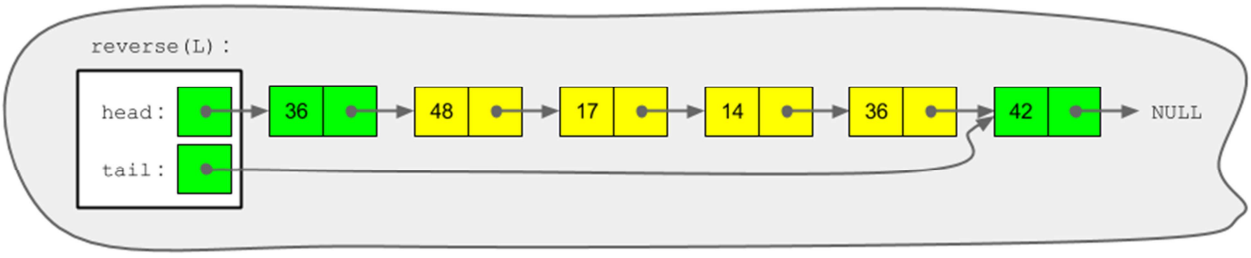


a) Complete the function `spliceinto(L, x, i)` which inserts the integer x into index location `i` of list L. If location `i` >= length of list L, then it appends node containing x at the end. If i < 0, then it inserts at location 0. For example:




```
// Post:  inserts node with data x into location i of list L
void spliceinto(LL_t * L, int x, int i) {
   . . .
}
```

b) Complete the function `reverse(L)` which *iteratively* reverses the list L. For example:
* Not for marks but good exam question: implement `reverse(L)` recursively

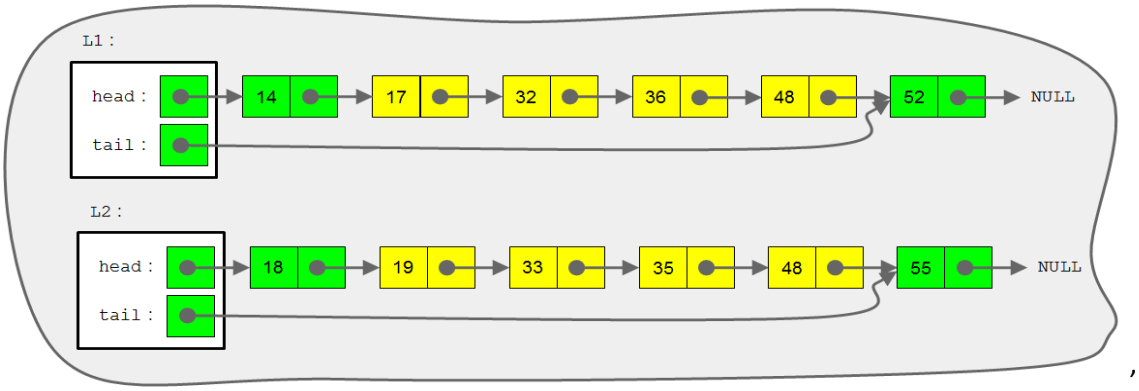



```
// Post:  nodes of list L are now reversed
void reverse(LL_t * L) {
   . . .
}
```

c) Complete the function `merge(L1, L2)` which is the merge function for two lists `L1` and `L2`. You may assume that `L1` and `L2` are in sorted order. Upon return, the list `L1` will contain the merged list, `L2` will be freed.

```
// Pre: L1 and L2 are ordered linked lists
// Post: L1 contains the merged (ordered) list of L1 and L2, L2 is freed
void merge(LL_t * L1, LL_t * L2) {
   . . .
}
```

For example, given



,