CS261: Midterm Exam 1

NAME: ____

INSTRUCTIONS

- This is a 45 minute, closed-book exam containing **TWO** problems.
- Cases of academic dishonesty will be filed to the Office of Student Conduct for disciplinary actions

Problem	Max points	Earned points
1.1	11	
1.2	29	
2.1	10	
2.2	8	
2.3	42	
TOTAL	100	

1. (40 points) Dynamic Arrays

For this problem, you will implement a function, `addDeque()`, for Deque implemented as a dynamic array. One of the input arguments of `addDeque()` is `flag` which indicates where to add the new element. For `flag == 0`, the new element should be added to the front of Deque; otherwise, for `flag != 0`, to the back of Deque. In your implementation, please feel free to use the C code provided for Problem 1. Please do not change the provided names of variables, functions, and data types. You cannot assume that any other code is available to you beyond the C code for Problem 1. In case you need additional C functions that are not provided, you will have to implement them.

1.1. (11 points) Pseudo Code of `addDeque()`

Briefly describe in words the main steps of your `addDeque()`.

Solution:

1. (0 points) – Check the validity of input parameters
2. (2 points) – Check if there is enough memory to add a new element; If not, call doubleCapacity() to allocate a larger block of memory to Deque
3. (1 point) If flag == 0, we add the new element to the front:
   (a) (2 points) Compute the new start index of Deque modulo capacity
   (b) (1 point) Assign the new element to Deque[start index]
4. (1 points) Else, we add the new element to the back:
   (a) (2 points) Compute the new back index of Deque modulo capacity
   (b) (1 point) Assign the new element to Deque[back index]
5. (1 point) Increment the size of Deque

1.2. (29 points) Write a C code for `addDeque()`

```
/* input: dq -- pointer to deque
          val -- value of the data element to be added
          flag -- flag == 0 => add to the front,
                  flag != 0 => add to the back
*/
void addDeque(struct deque *dq, TYPE val, int flag) {
    int backIndex;

   /* Check input arguments */
   assert(dq);

   /* (6 points)  Check memory capacity */
   if (dq->size == dq->capacity) _doubleCapacity(dq);

   /* (2 points)  Test the flag */
   if(flag == 0){  /* add to front */

       /* (6 points)  Decrement the front index modulo capacity */
       dq->start--;  if (dq->start < 0) dq->start += dq->capacity;

       /* or: dq->start = (dq->start - 1) % dq->capacity; */

       /* (2 points) Assign the new element  */
       dq->data[dq->start] = val;
   }
   /* (2 points)  Otherwise */
   else {   /* add to back */

        /* (6 points)  Compute the back index modulo capacity */
        backIndex = (dq->start + dq->size ) % dq->capacity;

        /* (2 points) Assign the new element  */
        dq->data[backIndex] = val;
   }
   /* (3 points)  Increment the size */
   d->size ++;
}
```

2. (60 points) Doubly-linked List Queue

For this problem, you will implement a maintenance function, `diffQueue()`, for Queue implemented as a doubly linked list. The function `diffQueue()` takes queues `q1` and `q2` as input arguments, and removes every element from `q1` that satisfies the following two conditions:

"Element is also present in `q2`"   AND   "Element is closer to the front of `q2` than to front of `q1`"

The resulting `q1` and `q2` should keep their initial FIFO properties.

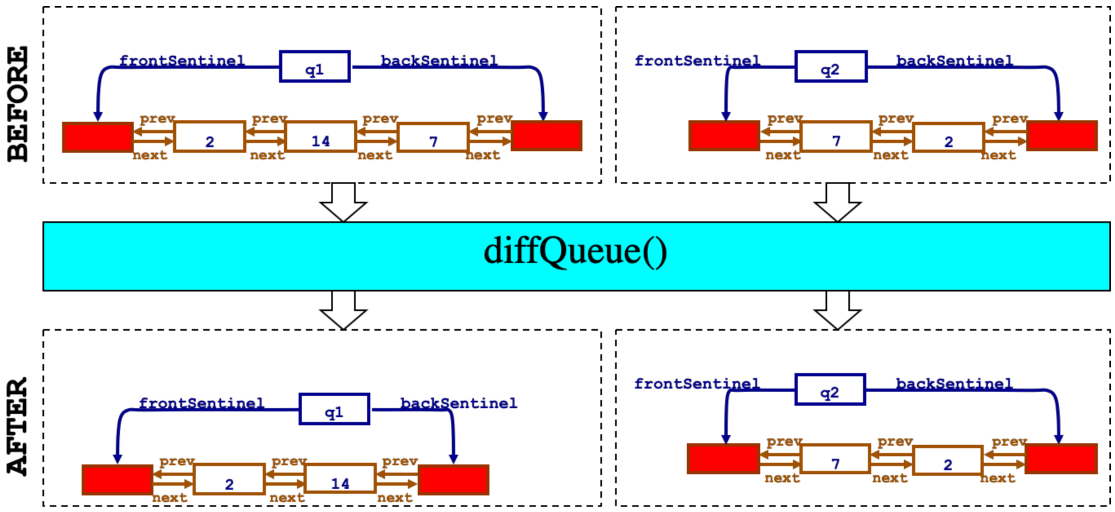

Figure 1: Example: the input queues `q1` and `q2` share two elements 2 and 7. After exiting `diffQueue()`, element 7 is removed from `q1`, but element 2 is not removed from `q1` because 2 in `q2` is *not* closer to the front than 2 in `q1`.

In your implementation, please feel free to use the C code provided for Problem 2. Please do not change the provided names of variables, functions, and data types. You cannot assume that any other code is available to you beyond the C code for Problem 2. In case you need additional C functions that are not provided, you will have to implement them.

2.1. (10 points) Pseudo Code of `diffQueue()`

Briefly describe in words the main steps of your `diffQueue()`.

Solution:

1. (0 points) Check the validity of input parameters
2. (1 point) Iterate over links of q2 using current2 until the back of q2
   (a) (1 point) Use counter2 to count how far current2 is from the front of q2
   (b) (1 point) Get the value val2 of the current link in q2
   (c) (1 point) Iterate over links of q1 using current1 until the back of q1
      i. (1 point) Use counter1 to count how far current1 is from the front of q1
      ii. (1 point) Get the value val1 of the current link in q1
      iii. (1 point) Check if EQ(current1->value,current2->value) AND (counter1 > counter2). If yes, remove the link pointed by current1 from q1:
         A. (1 point) Re-assign the pointers next and previous of the neighboring links of current1 in q1
         B. (1 point) Free current1 from memory
         C. (1 point1) Decrement the size of q1
   (d) End the iterations over links of q1
3. End the iterations over links of q2

2.2. (8 points) Based on your pseudocode, what is the time complexity of `diffQueue()` when `q1` and `q2` each has n elements?

Solution:

$O(n^2)$, where n is the number of elements in the queue.

2.3. (42 points) Write a C code for `diffQueue()`

Since the problem statement is not sufficiently precise, we will recognize solutions that estimate the relationship “closer” by considering either the current or original (input) queue q1.

Solution 1: This solution estimates the relationship “closer” relative to the current queue q1.

```
/*  Input:
      - q1: the first queue; elements should be removed from q1
      - q2: the second queue
    Post-condition: q1 and q2 preserve the FIFO property */

void diffQueue (struct dlist *q1, struct dlist *q2) {

   /* (0 points)  declare variables and check input arguments */
   struct dlink *current1;   /* points to the current link in q1 */
   struct dlink *current2;   /* points to the current link in q2 */
   int counter1;  /* counting how far the current link is from the front of q1 */
   int counter2;  /* counting how far the current link is from the front of q2 */
   assert(q1 && q2);

    /* (4 points) Initialize the variables before entering the loop over q2 */
    current2 = q2->frontSentinel; /* start iterations from the front of q2 */
    counter2 = 0;

    /* (4 points) Iterate over links in q2 */
    while (current2->next !=  q2->backSentinel){

       /* (4 points) Point to the first link in q2 */
       current2 = current2->next;
       counter2++;

       /* (4 points) Initialize the variables before entering the loop over q1 */
       current1 = q1->frontSentinel; /* start iterations from the front of q1 */
       counter1 = 0;

       /* (4 points) Iterate over links in q1 */
       while (current1->next !=  q1->backSentinel){

          /* (4 points) Point to the first link in q1 */
          current1 = current1->next;
          counter1++;

          /* (6 points) Check the two conditions for removing elements from q1 */
          if ( EQ(current1->value,current2->value) && counter1 > counter2 )

             /*  (12 points) remove the link pointed by current1 from q1 */
             /*  We call an auxiliary function, see the code below */
             _removeDLink (q1, current1);
}  }  }
```

Solution 2: This solution estimates "closer" relative to the original (input) queue q1. We first form an auxiliary array of links from q1 that need to be removed, and then remove them from q1.

```
void diffQueue (struct dlist *q1, struct dlist *q2) {
   /* (0 points)  declare variables and check input arguments */
   struct dlink *current1;   /* points to the current link in q1 */
   struct dlink *current2;   /* points to the current link in q2 */
   int counter1;  /* counting how far the current link is from the front of q1 */
   int counter2;  /* counting how far the current link is from the front of q2 */
   struct dlink *temp1[q1->size]; /* auxiliary array of links in q1 to be removed */
   int size_temp1 = 0; /* number of links in q1 to be removed */
   assert(q1 && q2);

    /* (4 points) Initialize the variables before entering the loop over q2 */
    current2 = q2->frontSentinel; /* start iterations from the front of q2 */
    counter2 = 0;

    /* (4 points) Iterate over links in q2 */
    while (current2->next !=  q2->backSentinel){

       /* (4 points) Point to the first link in q2 */
       current2 = current2->next;
       counter2++;

       /* (4 points) Initialize the variables before entering the loop over q1 */
       /* We can remove only links from q1 that come after the first counter2
             number of links from the front of q1 */
       counter1 = counter2;
       current1 = q1->frontSentinel; /* start iterations from the front of q1 */
       while (current1->next !=  q2->backSentinel && counter1){
            /* move forward along q1 */
            current1 = current1->next;
            counter1--;
       }

       /* (4 points) Iterate over links in q1 */
       while (current1->next !=  q1->backSentinel){

           /* (4 points) Point to the link in q1 that comes after the first
                        (counter2+1) number of links from the front of q1 */
           current1 = current1->next;

           /* (6 points) Check only EQ() for removing elements from q1 */
           if ( EQ(current1->value, current2->value) ){
              /* memorize the link to be removed from q1 */
              temp1[size_temp1] = current1;
              size_temp1++;
           }
        } /* end of q1 loop */
     }   /* end of q2 loop */
     /*  (12 points) remove the recorded links from q1 */
     while(size_temp){
        /*  We call an auxiliary function, see the code below */
        _removeDLink (q1, temp1[size_temp-1]);
        size_temp--;
     }
} /* end of function */




/* Auxiliary function for removing a given link from the queue
    Input:
       - pointer to the queue
       - pointer to the link to be removed from the queue
    Post-condition: the queue preserves the FIFO property */

void _removeDLink (struct dlist *queue, struct dlink *lnk){
   assert(queue && lnk);

    /*  (6 points) reconnecting the  neighboring links */
   lnk->previous->next = lnk->next;
   lnk->next->previous = lnk->previous;

   /*  (4 points) removing the link from memory */
   free(lnk);

  /*  (2 points) update the size of queue */
   queue->size--;
}
```

Code for Problem 1

```
#define TYPE  double
#define EQ(a,b) (a == b)

struct deque{
    TYPE *data;
    int size; /* number of elements */
    int capacity; /* memory capacity */
    int start; /* start index of deque  */
}

/* Initializes a deque */
void initDeque(struct deque * dq, int cap) {
   assert (dq && cap > 0);
   dq->capacity = cap;
   dq->size = dq->start = 0;
   dq->data = (TYPE *) malloc(dq->capacity * sizeof(TYPE));
   assert (dq->data != 0);
}


/* Doubles the memory capacity of a deque */
void _doubleCapacity (struct deque *dq) {
  assert (dq);
  int  j;
  TYPE * oldData = dq->data; /*memorize the old data*/
  int oldStart = dq->start; /*memorize the old start index */
  int oldSize = dq->size; /*memorize the old size*/
  int oldCapacity = dq->capacity; /*memorize the old capacity*/
  initDeque(dq, 2 * oldCapacity); /*new memory allocation*/
  for (j = 0 ; j < oldSize; j++) {/*copy back the old data*/
    dq->data[j] = oldData[oldStart++];
    if (oldStart >= oldCapacity) oldStart = 0;
  }
  free(oldData);
  dq->size = oldSize;
}
```

Code for Problem 2

```
#define TYPE double
#define EQ(a,b) (a==b)

struct dlink {
   TYPE value;
   struct dlink * next;
   struct dlink * previous;
};

struct dlist {
   int size;
   struct dlink * frontSentinel; /* front sentinel */
   struct dlink * backSentinel; /* back sentinel */
};

/* Initialize doubly linked list */
void initDList (struct dlist *dl) {
   assert(dq);
   dl->frontSentinel = (struct dlink *) malloc(sizeof(struct dlink));
   assert(dl->frontSentinel != 0);
   dl->backSentinel = (struct dlink *) malloc(sizeof(struct dlink));
   assert(dl->backSentinel);
   dl->frontSentinel->next = dl->backSentinel;
   dl->backSentinel->previous = dl->frontSentinel;
   dl->size = 0;
}
```