Data Structures

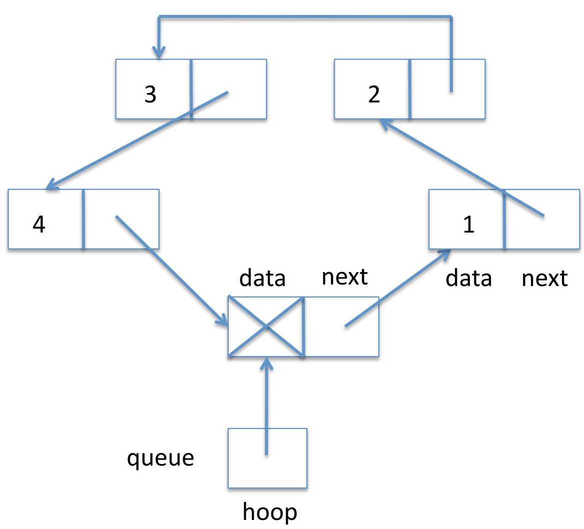
1 Queues and Linked Lists (from Midterm 1, Spring 2011)

Recall the definition of linked lists and the `is_segment` function that checks if a linked list beginning at `start` eventually arrives at `end`.

```
struct list_node {
  elem data;
  struct list_node* next;
};
typedef struct list_node list;

bool is_segment(list* start, list* end);
```

A *hoop* represents a queue as a cyclic linked list with one extra node. For example, a queue with items $1, 2, 3, 4$ (in this order, 1 at the front of the queue and 4 at the back) would be represented as the following hoop:



We define

```
struct queue_header {
  list* hoop;
};
typedef struct queue_header* queue;
```

As an example, here is a function that checks if the hoop is empty.

```
bool queue_empty(queue Q)
//@requires is_queue(Q);
{
  return Q->hoop == Q->hoop->next;
}
```

Task 1 (5 pts). Write a function that checks if a given queue is valid. [**Hint:** Use `is_segment`, and remember that pointers can be `NULL`.]

```
bool is_queue(queue Q) {

  if (Q == NULL) return false;
  if (Q->hoop == NULL) return false;
  return is_segment(Q->hoop->next, Q->hoop);

}
```

Task 2 (10 pts). Write a function to dequeue an element from the front of the queue. Your function should take constant time. [**Hint:** Draw a picture!]

```
elem deq(queue Q)
//@requires is_queue(Q);
//@requires !queue_empty(Q);
//@ensures is_queue(Q);
{

  elem e = Q->hoop->next->data;
  Q->hoop->next = Q->hoop->next->next;
  return e;

}
```

Task 3 (10 pts). Write a function to enqueue an element at the end of the queue. Your function should take constant time. [**Hint:** Draw a picture!]

```
void enq(queue Q, elem e)
//@requires is_queue(Q);
//@ensures is_queue(Q);
{

  list* h = alloc(list);
  h->next = Q->hoop->next;
  Q->hoop->data = e;
  Q->hoop->next = h;
  Q->hoop = h;
  return;

}
```

2 Stacks and Queues (from Midterm 1, Fall 2010)

Consider the following interface to stacks, as introduced in class.

```
struct stack_header {
  list* top;
  list* bottom;
};
typedef struct stack_header* stack;

stack stack_new();              /* O(1); create new, empty stack */
bool stack_empty(stack S);      /* O(1); check if stack is empty */
void push(stack S, elem e);     /* O(1); push element onto stack */
elem pop(stack S);              /* O(1); pop element from stack  */
```

In these problem you do not need to write annotations, but you are free to do so if you wish. You may assume that all function arguments of type `stack` are non-`NULL`.

Task 1 (10 pts). Write a function `rev(stack S, stack D)`. We require that D is originally empty. When `rev` returns, D should contain the elements of S in reverse order, and S should be empty.

```
void rev(stack S, stack D)
//@requires stack_empty(D);
//@ensures stack_empty(S);
{

  while (!stack_empty(S))
    push(D, pop(S));

}
```

Now we design a new representation of queues. A queue will be a pair of two stacks, `in` and `out`. We always add elements to `in` and always remove them from `out`. When necessary, we can reverse the `in` queue to obtain `out` by calling the function you wrote above.

```
struct queue_header {
  stack in;
  stack out;
};
typedef struct queue_header* queue;
```

Task 2 (10 pts). Write the `enq` function.

```
void enq(queue Q, elem x) {

  push(Q->in, x);

}
```

Task 3 (10 pts). Write the `deq` function. Make sure to abort the computation using an appropriate `assert()` statement if `deq` is called incorrectly.

```
int deq(queue Q) {

  assert(!stack_empty(Q->in) || !stack_empty(Q->out));

  if (stack_empty(Q->out)) {
    rev(Q->in, Q->out);
  }
  return pop(Q->out);

}
```

Now we analyze the complexity of this data structure. We are counting the total number of `push` and `pop` operations on the underlying stack.

Task 4 (10 pts). What is the worst-case complexity of a single `enq`? What is the worst-case complexity of a single `deq`? Phrase your answer in terms of big-O of m, where m is the total number of elements already in the queue.

$O(1)$ for `enq` and $O(m)$ for `deq`.