This handout presents two implementations of stacks, and three implementations of queues. Each implementation presents an identical stack or queue interface to the user, thereby providing a strong data abstraction barrier between the use of these data structures and their implementation.

1 Stacks

A *stack* is a Last–In, First–Out (LIFO) data storage structure. We will provide for our users a data type called `stack_type`, and an interface to stacks comprising four routines:

1. `stack_type create_stack(void)`: Creates a new stack and returns it.
2. `void push(stack_item_type item, stack_type stack)`: Adds the specified item to the top of the stack.
3. `stack_item_type pop(stack_type stack)`: Returns the top item on the stack, and, as a side–effect, deletes that item from the stack.
4. `BOOL empty_stack(stack_type stack)`: A predicate that returns `TRUE` if the stack is empty, and `FALSE` otherwise.

1.1 A Linked List Implementation

Here is a linked list implementation of stacks. Note that we represent a stack as a pointer to a linked list; this allows the routines that need to modify the stack passed to them to do so, even though the stack is passed by value.

```
typedef list_type *stack_type;
typedef list_item_type stack_item_type;

stack_type
create_stack(void)
{
  stack_type result;

  result = malloc(sizeof(list_type));
  assert(result != NULL);
  *result = the_empty_list;
  return(result);
}

void
push(stack_item_type item, stack_type stack)
{
  *stack = cons(item, *stack);
}
```

```
stack_item_type
pop(stack_type stack)
{
  stack_item_type result;

  assert(!empty_list(*stack));
  result = first(*stack);
  *stack = rest(*stack);
  return(result);
}

BOOL
empty_stack(stack_type stack)
{
  return(empty_list(*stack));
}
```

1.2 An Array Implementation

Here is an array implementation of stacks. Notice that the headers for the four functions are exactly the same as in the linked list implementation. This allows the array implementation to be substituted for the linked list implementation with no change in the code that uses stacks.

```
#define MAX_STACK_SIZE 128

typedef int stack_item_type;
typedef struct stack_tag *stack_type;
typedef struct stack_tag {
  int count;
  stack_item_type items[MAX_STACK_SIZE];
} stack_struct;

stack_type
create_stack(void)
{
  stack_type result;

  result = malloc(sizeof(stack_struct));
  assert(result != NULL);
  result->count = 0;
  return(result);
}

void
push(stack_item_type item, stack_type stack)
{
  assert(stack->count < MAX_STACK_SIZE);
  stack->items[stack->count] = item;
  stack->count++;
}

stack_item_type
pop(stack_type stack)
{
  assert(stack->count > 0);
  stack->count--;
  return(stack->items[stack->count]);
}
```

```
BOOL
empty_stack(stack_type stack)
{
  return(stack->count == 0);
}
```

2 Queues

A *queue* is a First–In, First–Out (FIFO) data storage structure. We will provide for our users a data type called `queue_type`, and an interface to queues comprising four routines:

1. `queue_type create_queue(void)`: Creates a new queue and returns it.
2. `void enqueue(queue_item_type item, queue_type queue)`: Adds the specified item to the rear of the queue.
3. `queue_item_type dequeue(queue_type queue)`: Returns the item at the front of the queue, and as a side–effect, deletes that item from the queue.
4. `BOOL empty_queue(queue_type queue)`: A predicate that returns `TRUE` if the queue is empty, and `FALSE` otherwise.

2.1 A Linked List Implementation

Here is a linked list implementation of queues. It requires the ability to add an item to the end of a linked list, which is accomplished via the `append1` function. Notice that linked lists are used without any knowledge of how they are actually implemented.

```
list_type
append1(list_type list, list_item_type item)
{
  if (empty_list(list))
    return(cons(item, the_empty_list));
  else return(cons(first(list), append1(rest(list), item)));
}

typedef list_type *queue_type;
typedef list_item_type queue_item_type;

queue_type
create_queue(void)
{
  queue_type result;

  result = malloc(sizeof(list_type));
  assert(result != NULL);
  *result = the_empty_list;
  return(result);
}
```

```
void
enqueue(queue_item_type item, queue_type q)
{
  *q = append1(*q, item);
}

queue_item_type
dequeue(queue_type q)
{
  queue_item_type result;

  assert(!empty_list(*q));
  result = first(*q);
  *q = rest(*q);
  return(result);
}

BOOL
empty_queue(queue_type q)
{
  return(empty_list(*q));
}
```

2.2 A Pointer Implementation

Here is a pointer implementation of queues. A queue is defined to be a pointer to a struct that contains the head and the tail of a queue (each of which is a pointer to a linked list–like structure). This implementation allows memory to be properly freed when it is no longer needed.

```
typedef int queue_item_type;
typedef struct queue_node_struct {
  queue_item_type data;
  struct queue_node_struct *next;
} *queue_node_ptr;
typedef struct queue_tag *queue_type;
typedef struct queue_tag {
  queue_node_ptr front;
  queue_node_ptr rear;
} queue_struct;

queue_type
create_queue(void)
{
  queue_type result;

  result = malloc(sizeof(queue_struct));
  assert(result != NULL);
  result->front = NULL;  /* This indicates an empty queue. */
  result->rear = NULL;   /* Just for neatness */
  return(result);
}
```

```
void
enqueue(queue_item_type item, queue_type q)
{
  queue_node_ptr temp;

  temp = malloc(sizeof(struct queue_node_struct));
  assert(temp != NULL);
  temp->data = item;
  temp->next = NULL;
  if (q->front == NULL)
    q->front = temp;
  else q->rear->next = temp;
  q->rear = temp;
}

queue_item_type
dequeue(queue_type q)
{
  queue_node_ptr temp;
  queue_item_type result;

  assert(q->front != NULL);
  temp = q->front;
  q->front = q->front->next;
  result = temp->data;
  if (q->front == NULL)
    q->rear = NULL;    /* Again, just for neatness. */
  free(temp);          /* Free up the space used by the removed item. */
  return(result);
}

BOOL
empty_queue(queue_type q)
{
  return(q->front == NULL);
}
```

2.3 An Array Implementation

Here is an array implementation of queues. It uses the C modulus operator (`%`) to wrap around the end of an array when a subscript to the array becomes too large.

```
#define MAX_QUEUE_SIZE 128

typedef int queue_item_type;
typedef queue_item_type queue_array[MAX_QUEUE_SIZE];
typedef struct queue_tag *queue_type;
typedef struct queue_tag {
  queue_array items;
  int front;
  int count;
} queue_struct;
```

```
queue_type
create_queue(void)
{
  queue_type result;

  result = malloc(sizeof(queue_struct));
  assert(result != NULL);
  result->front = 0;
  result->count = 0;
  return(result);
}

void
enqueue(queue_item_type item, queue_type q)
{
  assert(q->count < MAX_QUEUE_SIZE);
  q->items[(q->front + q->count) % MAX_QUEUE_SIZE] = item;
  q->count++;
}

queue_item_type
dequeue(queue_type q)
{
  queue_item_type result;

  assert(q->count > 0);
  result = q->items[q->front];
  q->front = (q->front + 1) % MAX_QUEUE_SIZE;
  q->count--;
  return(result);
}

BOOL
empty_queue(queue_type q)
{
  return(q->count == 0);
}
```

The code presented in this handout is available in the directory ~`mayfield` in the files `queues-array.c`, `queues-list.c`, `queues-pointer.c`, `stacks-array.c` and `stacks-list.c`. To use these routines with your own code, you can access the versions that are part of the CMSC 202 library; see the files `~mayfield/lib202/stacks.h` and `~mayfield/lib202/queues.h` for details.

3 Exercises

1. Write a program that reads in numbers from the standard input until end–of–file, then prints the numbers in reverse order. Your program should use a stack to reverse the order of the numbers.

2. Write the function `copy_queue` that takes a `queue_type` as its argument and returns another `queue_type` as its result. The return queue must contain all of the items in the original queue in the same order that they were found in the original queue. The original queue must end up unchanged. Your function must lie above the queue abstraction barrier (*i.e.*, all manipulation of queues must be done using `create_queue`, `enqueue`, `dequeue`, and `empty_queue`).