Stack: Data structure that supports two main operations that allows elements to be pushed onto the stack, or popped off of the stack

+ Uses a last-in, first-out abstraction, i.e. the last element pushed on the stack will be the first element popped from the stack.

* Stacks can be efficiently implemented as a singly-linked list.

① Use a typedef to directly map the List datatype to a stack datatype

```
typedef List Stack;
```

② Define functions that will operate on the stack. To avoid confusion and to provide a proper abstraction for interfacing with a stack, we will define a set of functions (i.e. interface) for handling stacks.

```
void stack_init (Stack * stack) {
    list_init (stack);
}
void stack_destroy (Stack * stack) {
    list_destroy (stack);
}
int stack_size (Stack * stack) {
    return list_size (stack);
}
```

```
int stack_push (stack *stack, int val) {
    return list_ins_next(stack, NULL, val);
}
```

* Pushing onto stack can be implemented as inserting an element at the head of the list

```
int stack_pop (stack *stack, int *val) {
    if (stack->head != NULL) *val = stack->head->val;
    return list_rem_next(stack, NULL);
}
```

* Popping from stack can be implemented by copying the data value from the head of the list and removing the head of the list

```
void stack_peek (stack *stack, int *val) {
    if (stack->head != NULL) *val = stack->head->val;
}
```

* Peek allows one to view the value at the top of the stack, without removing it

Queues: Data structures that supports two main operations allowing data to be inserted at the end of the queue and removed from the head of the queue

+ Uses a first-in, first-out abstraction

* Queues can be efficiently implemented using singly linked list.

① Use typedef to map the DList data type to a Queue datatype

```
typedef List Queue;
```

② Define functions that operate on Queue abstraction

```
void queue_init(Queue *q) {
    list_init(q);
}

void queue_destroy(Queue *q) {
    list_destroy(q);
}
int queue_size(Queue *q) {
    return list_size(q);
}
```

```
int queue_enqueue(Queue *q, int val) {
    return list_ins_next(q, list_tail(q), val);
}
```

* Enqueue can be implemented as inserting element after the tail of the list

```
int queue_dequeue(Queue *q, int *val) {
    if (q->head != NULL) *val = q->head->val;
    return list_rem_next(q, NULL);
}
```

* Dequeue can be implemented by copying data at head of list and removing the head

```
void queue_peek(Queue *q, int *val) {
    if (q->head != NULL) *val = q->head->val;
}
```

* Peak allows you to view element at front of queue without removing it