Last lecture

- Types of Programming Languages
  - Procedural/Imperative (C, Pascal, Fortran)
  - Functional Languages (Haskell, Lisp)
  - Object-Oriented (C++, Java)
Two Special List Structures (in C)

Stack
- Last-in First-out (lifo)
- Key operations are push, pop, top and empty

Queue
- First-in First-out (fifo)
- Key operations are enqueue, dequeue, front, empty

Both Stack and Queue are simpler to implement than a general linked list
struct for Stacks

Each “cell” of the structure carries the following two things:

- A piece of data
- A pointer to the “cell” below it in the stack

We can “package” this as a recursive struct declaration.

```c
struct elem {
    /* structure of an element on the */
    data d;   /* stack: just one piece of data, and */
    struct elem *next; /* pointer to ’next’ cell */
};
typedef struct elem elem;
```
#define EMPTY 0
#define FULL 10000
typedef char data;
typedef enum {false, true} boolean;

struct elem { /* structure of an element on the stack: */
  data d; /* just one piece of data, and pointer */
  struct elem *next; /* to 'next' cell */
};
typedef struct elem elem;

typedef struct { /* Stack is just one element (the 'top' */
  int count; /* one) plus also a count of items in */
  elem *top; /* entire Stack. */
} stack;
void initialize(stack *stk);
void push(data d, stack *stk);
data pop(stack *stk);
data top(const stack *stk);
boolean empty(const stack *stk);
boolean full(const stack *stk);
Memory allocation

No idea how many items we will get (range 0 to 10000)

- Allocate memory on a **cell-by-cell** basis with `malloc`
- free the space used by a cell whenever it is ‘pop’-ed

```c
malloc free
```
push - stack is growing

**pre-requisite:** Must know that stack is NOT “full”.

```c
void push(data d, stack *stk) {
    elem *p;

    p = malloc(sizeof(elem));
    (*p).d = d;
    (*p).next = (*stk).top;
    (*stk).top = p;
    (*stk).count++;
}
```
Precedence of * vs .

Our push function (and other functions) takes as parameter a pointer to a stack structure.

The . operator (used to access a part of a struct) has higher precedence than the * operator (used to de-reference a pointer). For this reason, in the code above,

we need the parentheses in statements

\[
(*p).d = d;,
\]

\[
(*p).next = (*stk).top;
\]

\[
\ldots
\]

So we usually use C’s abbreviation -> :

\[
x->y \text{ means } (*x).y
\]

And if you’re doing ‘pointer chasing’ (often considered bad style),

\[
x->y->z \text{ means } (x->y)->z \text{ means } (**x).y.z
\]
void push(data d, stack *stk) {
    elem *p;
    p = malloc(sizeof(elem));
    p->d = d;
    p->next = stk->top;
    stk->top = p;
    stk->count++;
}
pop - stack is shrinking

What happens if we pop an empty stack? Chaos and despair! So we’ll use assert to crash the program right now. User should check with empty before calling pop.

data pop(stack *stk) {
    data d;
    elem *p;

    assert(stk->count > 0);
    p = stk->top;
    d = p->d;
    stk->top = p->next;
    stk->count--;
    free(p);
    return d;
}
void initialize(stack *stk) {
    stk->count = 0;
    stk->top = NULL;
}
top, empty, full operations

/* User should check stack is not empty */
data top(const stack *stk) {
    assert(stk->count > 0);
    return stk->top->d;
}

boolean empty(const stack *stk) {
    return ((boolean) (stk->count == EMPTY));
}

boolean full(const stack *stk) {
    return ((boolean) (stk->count == FULL));
}
Example of applying Stack

Reverse a string

- Go through the string, pushing each character onto the stack.
- Now pop each item off the stack, directly onto standard output.

DEMO!!!
typedef struct { /* Queue has pointers to two 'elems's: */
    int cnt; /* the 'front' where items are taken off, */
    elem *front; /* and 'rear' where items put on. */
    elem *rear;
} queue;

void initialize(queue *q);
void enqueue(data d, queue *q);
data dequeue(queue *q);
data front(const queue *q);
boolean empty(const queue *q);
boolean full(const queue *stk);
Summary

- Rules of Stacks and Queues
- Implementation of Stacks
- Application of Stacks - reversing a string
- Go to course webpage for code (Stacks and Queues).

THURSDAY – we start REVISION!!!!