Assignment 4

- Assignment 4 will be released later today
- It is the most difficult assignment this quarter
- It is the longest assignment this quarter
- Suggested schedule
  - Work on problems 1, 2, 3, 4, and 6 before Monday
  - Focus on the midterm next week
  - Finish the assignment after the midterm
- This assignment will give you great programming experience! You will see the difference.
Where We Are

- We have seen
  - The concept of a struct
  - Dynamic memory allocation (malloc/free)

- Given these two concepts, we can now create dynamic data structures
  - Structures whose size grows and shrinks during program execution
  - Concrete examples today: stack in class
    - (and queue on your own)
  - You will create a list and a tree in assignment 4
Our program is longer today, so we will split it into two modules: `stack` and `main-stack`

- Such a split will also allow us to reuse the stack module in different programs

Overall, we will have three files

- `stack.c`: Functions that implement the stack
  - push, pop, is_empty, and print
- `stack.h`: All the function prototypes
- `main-stack.c`: A program that uses the stack
  - Must include `stack.h`
Self-Referential Structures

- Contains a pointer to a struct of the same type

```c
typedef struct node {
    int value;
    struct node *next;
} Node;
```

- Can contain more than one pointer
  - Example: a double-linked list will have 2 pointers
- These pointers are called links
- Typical building block for data structures
- Let's build a stack and, on your own, a queue...
Node *top;
Pointer to a Node structure

One Node structure

Node.value
Node.next
NULL
Push an Element onto the Stack

```
Node.value

Node.next

NULL

```
Push an Element onto the Stack

Node.next
Node.value

NULL
Pop an Element from the Stack

```
|   |   |   |
X | 4 | 3 |
  |   |   |
```

```
|   |   |   |
2 |   | 1 |
  |   |   |
```

```
|   |   |   |
1 |   | 1 |
  |   |   |
```

```
|   |   |   |
   |   |   |
```

Node.value

Node.next

NULL
Writing the Stack Module

- Now that we know how a stack works, let's take a look at the corresponding C code
void print(Node *top) {
    Node *current = top;
    while ( current != NULL ) {
        printf("%d\n", current->value);
        current = current->next;
    }
}
Create a New Stack

- Initializing stack: Node *top = NULL;

Diagram:

```
  top
  ↓
  [Node]
```

```c
Node *top = NULL;
```
// Client code
Node *top = NULL;
int i = 3;
push(&top, i);

How should we implement the push function?
Push First Data Item Onto Stack

- **Step 0:** Initial state
  - Empty stack

- **Step 1:** Allocate space for a new element
  - New element

- **Step 2:** Update pointers to add element to stack
  - New stack
Push Subsequent Data Item Onto Stack

- **Step 0:** Initial state

- **Step 1:** Allocate space for a new element

- **Step 2:** Update pointers to add element to stack
The “push” Function

```c
void push(Node **top, int value) {
    Node *e = (Node*)malloc(sizeof(Node));
    if (!e) {
        fprintf(stderr,"Out of memory\n");
        return;
    }
    e->value = value;
    e->next = *top;
    *top = e;
}
```
Popping Data From Stack

// Client code
Node *top = NULL;
push(&top, 1);
push(&top, 2);
push(&top, 3);
...
int value = pop(&top)

How should we implement the `pop` function?
Popping Data From Stack

- Pop an element from stack

Step 1

Step 2: deallocate
int pop(Node **top) {
    if ( ! is_empty(*top) ) {
        Node *removed = *top;
        int value = removed->value;
        *top = removed->next;
        free(removed);
        return value;
    }
    return -1;
}
Other Data Structures

- Other data structures in C can be implemented in a similar manner

- **Self-referential structures form the basic elements**

- **When inserting**
  - Allocate space for new element (malloc)
  - Initialize its fields
  - Update pointers

- **When removing**
  - Update pointers
  - Reclame space used by deleted element (free)
The following slides show another data structure: the queue
You can find the code for that example in queue.c, queue.h, main-queue.c
Second Example: Queue

- This time we need to keep around two pointers
  - **head**: pointer to the head of the queue
  - **tail**: pointer to the end of the queue
- Enqueue a value: value = 4
- Step 1: Allocate memory for new element and initialize fields
Step 2: Update links to add element to the end of the queue
Enqueue Operation

- Special case: adding first element to an empty queue
Enqueue Operation

- Special case: adding first element to an empty queue
Elements are removed from the head of the queue
Dequeue Operation

- Step 1: Update links
- Step 2: Deallocate element
Dequeue Operation

- Special case: removing the last element from a queue

Source code is in:
- queue.h queue.c, main-queue.c
Summary

- Quite easy to build useful structures

- Be systematic
  - One method allocates new elements
    - Example: enqueue, push
  - One method deallocates elements
    - Example: dequeue, pop

- Be careful
  - Watch-out for corner cases (ex: empty queue)
Frequent Bugs

- **Memory leak**: forgetting to free memory
  - Example: remove element from list, forget to free it, and lose all pointers to that element

- **Dangling pointers**
  - Can cause crash
  - Can cause you to overwrite other data

- **Good news**: tools exist to help you catch these bugs: *dmalloc, valgrind* (we will not have time to cover these tools in class)
Readings

- No additional readings for this class
- Examine the examples carefully
  - Pay attention to the parameters
  - Either Node * (pointer to a Node)
  - Or Node** (pointer to a pointer to a Node)