
COMP1511 - Programming Fundamentals

— Week 9 - Lecture 16 —

What are we covering today?

Finishing our Queue Implementation

- Destroying and Freeing
- Returning the number of items

Another Abstract Data Type

- Stacks
- Implementing with other data structures

Recap - Abstract Data Types

Making our own types with specific uses

- Declare our functionality in a Header (*.h) file
- Hide our Implementation in a *.c file
- The Header declares the type and the functions
- All the implementation is left out of the header
- The C file defines the underlying implementation

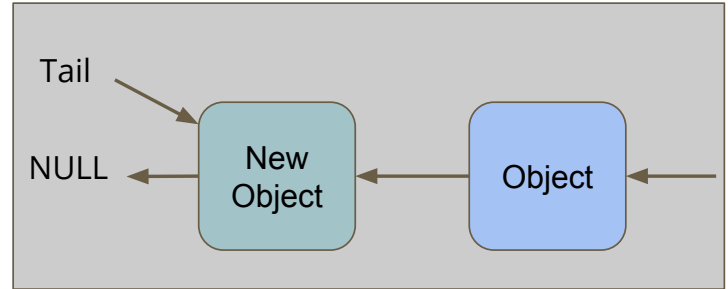
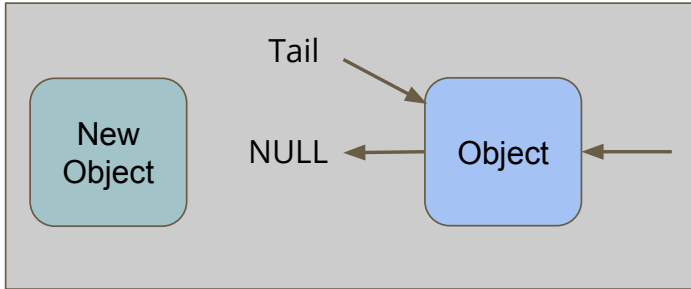
Finishing our Queue

Continuing the example

- We need a main file to use the Queue
- Let's try adding and removing
- We're not cleaning our memory properly yet
- So we need a function that frees an entire queue
- Also, a function that returns how many items are in the queue
- This makes it easier for someone to use without risking errors

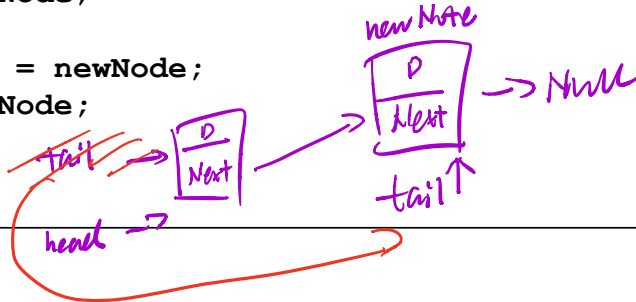
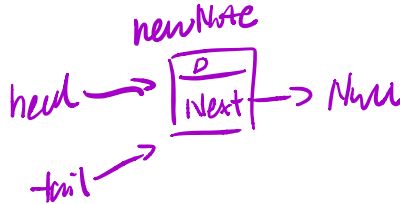
Adding to the tail

- Connect the new object to the current tail
- Move the tail pointer to the new last object
- We no longer need to loop through the whole queue to find the tail



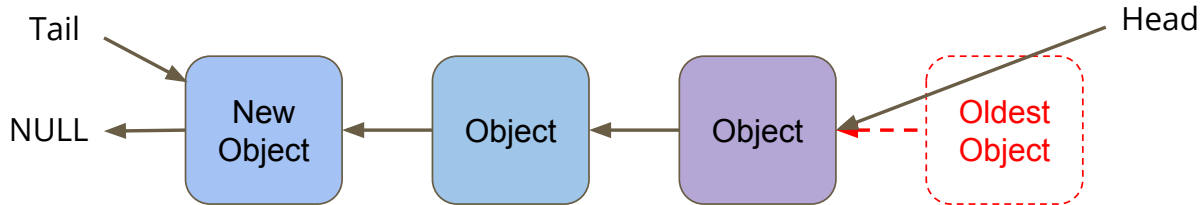
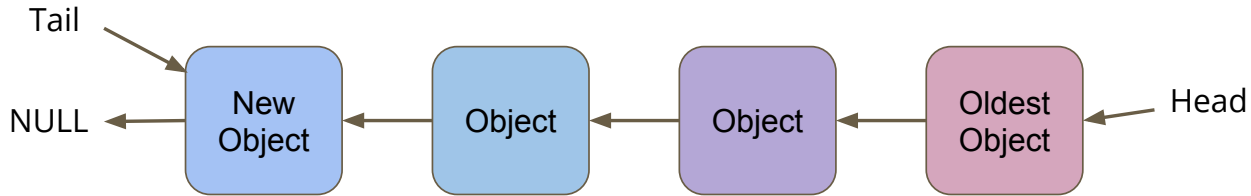
Code for Adding

```
void queueAdd(Queue q, int item) {  
    struct queueNode *newNode = malloc(sizeof(struct queueNode));  
    newNode->data = item;  
    newNode->next = NULL;  
  
    if (q->tail == NULL) {  
        // Queue is empty  
        q->head = newNode;  
        q->tail = newNode;  
    } else {  
        q->tail->next = newNode;  
        q->tail = newNode;  
    }  
}
```



Removing a Node

The only node that can be removed is the head (the oldest node)

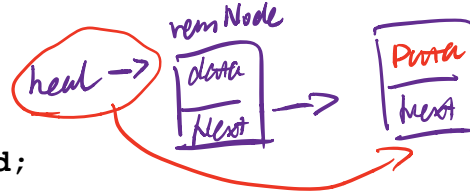


Code for Removing

```
// Remove the head from the list and free the memory used
int queueRemove(Queue q) {
    if (q->head == NULL) {
        printf("Attempt to remove an element from an empty queue.\n");
        exit(1);
    }
    // Keep track of the old head
    int returnData = q->head->data;
    struct queueNode *remNode = q->head;

    // move the queue to the new head and free the old
    q->head = q->head->next;
    free(remNode);

    return returnData;
}
```



Testing Code in our Main.c

```
int main(void) {  
    printf("Creating the Queue for Ice Cream.\n");  
    Queue iceQueue = queueCreate();  
    int id = 1;  
    printf("Person %d joins the queue!\n", id);  
    queueAdd(iceQueue, id);  
    id = 2;  
    printf("Person %d joins the queue!\n", id);  
    queueAdd(iceQueue, id);  
    id = 3;  
    printf("Person %d joins the queue!\n", id);  
    queueAdd(iceQueue, id);  
  
    printf("Person %d just got their ice cream!\n", queueRemove(iceQueue));  
    printf("Person %d just got their ice cream!\n", queueRemove(iceQueue));  
    printf("Person %d just got their ice cream!\n", queueRemove(iceQueue));  
    return 0;  
}
```

remove head

Our queue.h Header File

```
// queue type hides the struct that it is
// implemented as
typedef struct queueInternals *Queue;

// functions to create and destroy queues
Queue queueCreate(void);
void queueFree(Queue q);

// Add and remove items from queues
// Removing the item returns the item for use
void queueAdd(Queue q, int item);
int queueRemove(Queue q);

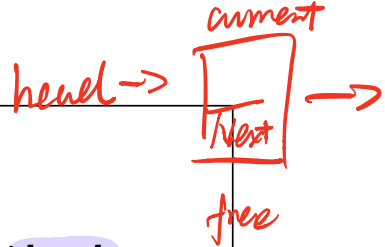
// Check on the size of the queue
int queueSize(Queue q);
```

queueFree()

Free all the memory in the linked list that we're using

- Loop through the list
- free() each node as we go

```
// Destroy and Free the entire queue
void queueFree(Queue q) {
    while (q->head != NULL) {
        struct queueNode *current = q->head;
        q->head = q->head->next;
        free(current);
    }
}
```



Testing for memory leaks

Let's use `gcc --leakcheck`

```
int main(void) {  
    Queue iceQueue = queueCreate();  
    queueAdd(iceQueue, 1);  
    queueAdd(iceQueue, 2);  
    queueAdd(iceQueue, 3);  
  
    queueFree(iceQueue);  
}
```

- What happens when we run with memory leak checking?
- Remember that all memory allocated with `malloc()` must be freed!

queueFree() Improved

Remember to free all the memory allocations!

```
// Destroy and Free the entire queue
void queueFree(Queue q) {
    while (q->head != NULL) {
        struct queueNode *current = q->head;
        q->head = q->head->next;
        free(current);
    }
    free(q);
}
```

↳ free tail and head.

Number of items in the Queue

Our last function is `queueSize()`

- Loop through the list until the end
- Count how many elements are in it

```
// Return the number of items in the queue
int queueSize(Queue q) {
    struct queueNode *iterator = q->head;
    int counter = 0;
    while(iterator != NULL) {
        counter++;
        iterator = iterator->next;
    }
    return counter;
}
```

Can we be trickier?

Maybe we don't want to loop through the whole list every time?

- We have a `queueInternals` struct that can store information
- How about we store the `size` there?

```
// Queue internals holds a pointer to the start of a linked list
struct queueInternals {
    struct queueNode *head;
    struct queueNode *tail;
    int size;
};
```

- Then, whenever we add or remove a node, we add or subtract 1 from this variable

Completing our Queue

To go along with our size variable ...

- queueCreate will set the size to 0 *0*
- queueAdd will add 1 *+1 size++;*
- queueRemove will subtract 1 *-1 size--;*

In our testing main(), we can now show this working with a loop:

```
while(queueSize(iceQueue) > 0) {  
    printf("Person %d just got their ice cream!\n", queueRemove(iceQueue));  
    printf("There are %d people in the queue.\n", queueSize(iceQueue));  
}
```


More thoughts on the Queue

Whatever includes the queue only sees the header

- When we're using ADTs we don't know (or need to know) the implementation
- What if this queue had been implemented using an array?

Challenge

- Implement queue.c using an array instead of a Linked List
- There are several different ways to make that work!

堆叠

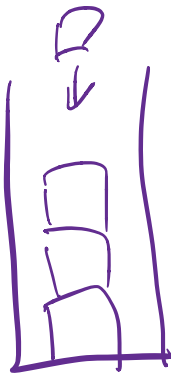
Stacks - another Abstract Data Type

A stack is a very common data structure in programming

- It is a "Last in first out" structure
- You can put something on top of a stack
- You can take something off the top of a stack
- You can't access anything underneath

访问

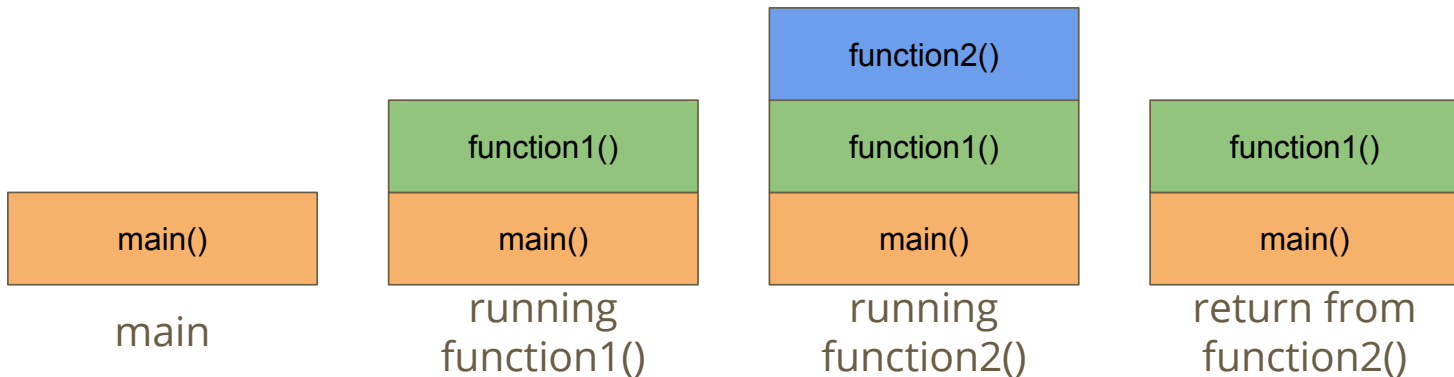
下面



This is actually how functions work!

The currently running code is on the top of the stack

- `main()` calls `function1()` - only `function1()` is accessible
- `function1()` calls `function2()` - only `function2()` is accessible
- control returns to `function1()` when `function2()` returns



What kind of functions does a stack need?

Functionality to put in a header file

- create
 - free
 - push (add to the top of the stack)
 - pop (remove from the top of the stack)
 - top (show the top without removing it)
 - size
- word Free (stack s) {
free (s); }*

We'll only have time for some of these today

A Stack Header

Looks eerily familiar to Queue...

```
// stack type hides the struct that it is implemented as
typedef struct stackInternals *Stack;

// functions to create and destroy stacks
stack stackCreate(void);
void stackFree(Stack s);

// Push and Pop items from stacks
// Removing the item returns the item for use
void stackPush(Stack s, int item); → add
int stackPop(Stack s);

// Check on the size of the queue
int stackSize(Stack s);
```

Implementation

What is our internal data structure going to be?

- We could use a linked list again
- We could use an array
- Whichever it is, it should be invisible to whoever includes the `stack.h` file
- For this example, let's use an array (just for a change)
- Our data will be stored in an array with a large maximum size
- We'll keep track of where the top is with an int

stack.c

```
// Struct representing the stack using an array
struct stackInternals {
    int stack[MAX_STACK_SIZE]; array
    int top; stackData
};

// create a new stack
stack stackCreate() {
    stack s = malloc(sizeof(struct stackInternals));
    s->top = 0;
    return s;
}
```

= struct stackInternals *s

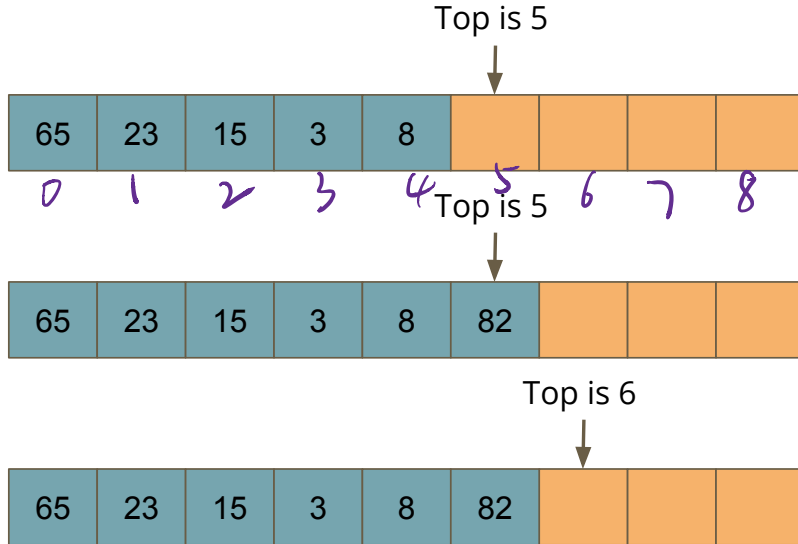
Push and Pop

These should only interact with the top of the stack

- **Push** should add an element after the end of the stack
- It should then move the top index to that new element
- **Pop** should return the element on the top of the stack
- It should then move the top index down one

Push

Push a new element "82" onto the stack



The stack starts like this

82 is added at top's index

Top then moves up one

Push code

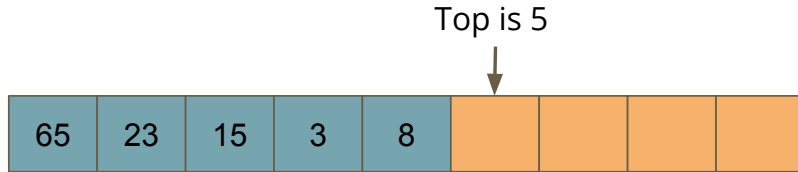
```
// Add an element to the top of the stack
void stackPush(stack s, int item) {
    // check to see if we've used up all our memory
    if(s->top == MAX_STACK_SIZE) {
        printf("Maximum stack size reached, cannot push.\n");
        exit(1);
    }
    s->stackData[s->top] = item;
    s->top++;
}
```

满格子

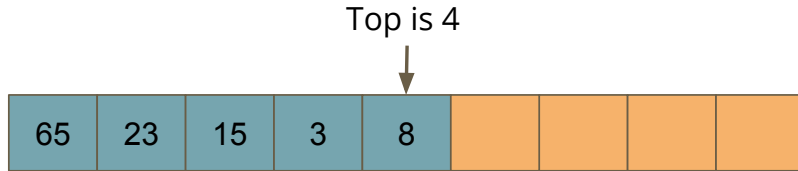
↑ array 在 stack internal 出现过

Pop

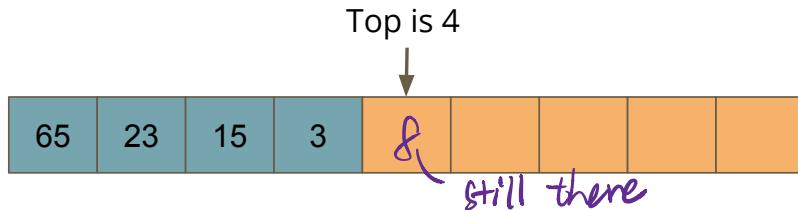
Pop removes the top element from the stack



The stack starts like this



Top moves down one



Read the element at top and return it

Pop code

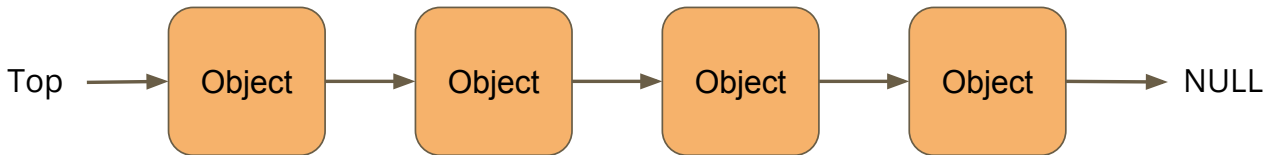
```
// Remove an element from the top of the stack
int stackPop(stack s) {
    // check to see if the stack is empty
    if(s->top <= 0) {
        printf("Stack is empty, cannot pop.\n");
        exit(1);
    }
    s->top--;
    return s->stackData[s->top];
}
```

只到top, not actually free integer

What if this were a linked list?

Implementation should be invisible to the including code

- Let's try to implement the same functions with a linked list
- We'll add elements to the end
- We'll also remove elements from the same end



Linked List Implementation

```
struct stackInternals {  
    struct node *top;  
};  
struct node {  
    struct node *next;  
    int data;  
};  
  
stack stackCreate() {  
    stack s = malloc(sizeof(struct stackInternals));  
    s->top = NULL;  
    return s;  
}
```

top → Null

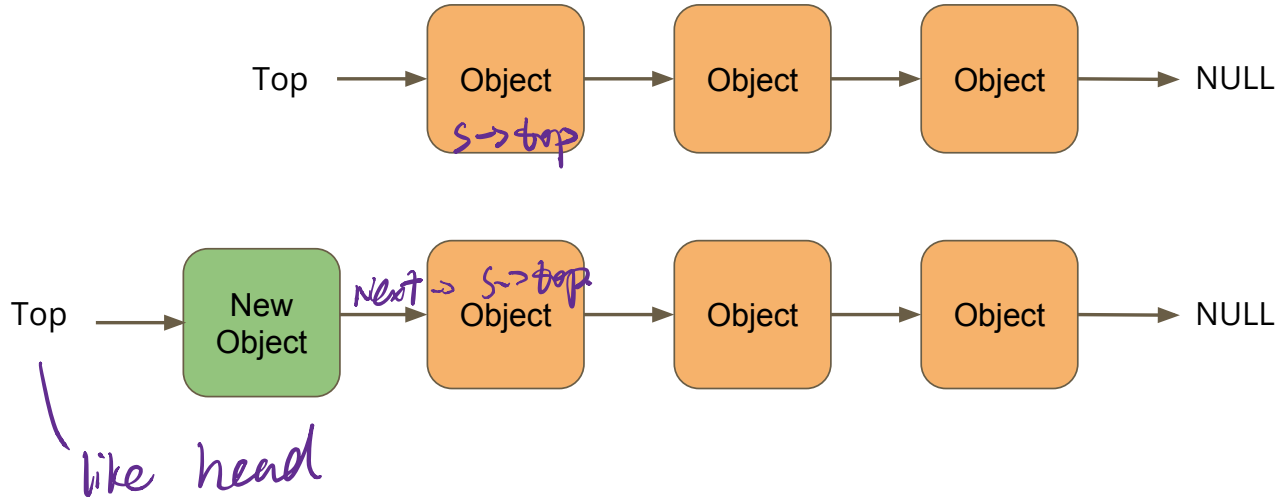
Push and Pop with a Linked List

All of our changes will apply to the top of the list

- **Push** adds an element to the top of the list
- Top will then point at that element
- **Pop** removes the top element of the list and returns it
- Top will then point at the next element

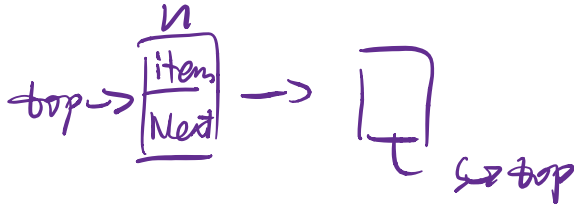
Push

Add a node to the top of the list



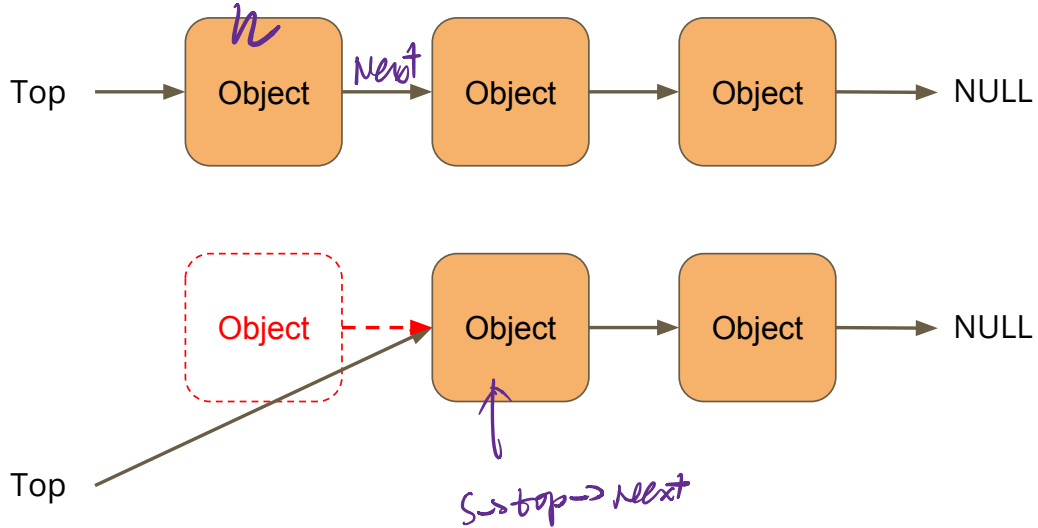
Push Code

```
// Add an element on top of the stack
void stackPush(stack s, int item) {
    struct node *n = malloc(sizeof (struct node));
    if (n == NULL) {
        printf("Cannot allocate memory for a node.\n");
        exit(1);
    }
    n->data = item;
    n->next = s->top;
    s->top = n;
}
```



Pop

Remove the node from the top of the list



Pop code

```
// Remove the top element from the stack
int stackPop(stack s) {
    if(s->top == NULL) { top → Null
        printf("Stack is empty, cannot pop.\n");
        exit(1);
    }
    // keep a pointer to the node so we can free it
    struct node *n = s->top;
    int item = n->data;
    s->top = s->top->next;
    free(n);
    return item;
}
```

Hidden Implementations

Neither Implementation needs to change the Header

- The main function doesn't know the difference!
- The structures and implementations are hidden from the header file and the rest of the code that uses it
- If we want or need to, we can change the underlying implementation without affecting the main code