Data Structures in C

Prof. Georg Feil

Abstract Data Types – Stacks & Queues

Summer 2018

Acknowledgement

- These lecture slides are partly based on slides by Professor Simon Hood
- Additional sources are cited separately

Reading/Video Assignment (required)

- Please watch this YouTube video
  - https://www.youtube.com/watch?v=92S4zgXN17o&index=1&list=PL2_aWCzGMAwI3W_JlcBbtYTwiQSsOTa6P

- Also read Data Structures (recommended textbook)
  - Chapter 5

(Note the textbook does a few things we would consider poor style, for example one-letter variable names and using int for Boolean values.)

The Abstract Data Type (ADT)

- An abstract data type is a description of a set (type) of arbitrary data values and their associated operations
  - The ADT does not make any assumptions about how the data is represented or stored in the computer
  - Think of it as a “black box” that provides some functions to manage a collection of data
- For example, a list of integers that provides a way to add() new numbers and remove() existing numbers is an ADT
- An abstract data type does not have a specific implementation or internal data format until it’s implemented as a specific concrete data structure
  - e.g. an array of integers is a concrete data structure. We know how it’s stored in memory, how operations work, and their speed

ADT benefits

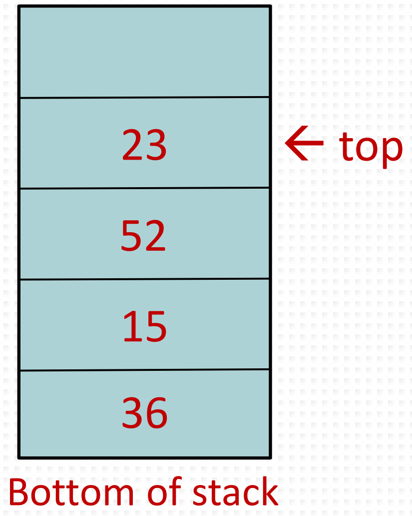
- An important benefit of an ADT is that we can change the internal implementation at any time
  - The code which uses the list – in our example calls add() or remove() – doesn’t need to change
  - We don’t know how fast or slow each operation is until we implement it in a specific way
  - We can substitute an “improved” version of the ADT any time
- Abstract data types can also be flexible regarding what kind of data they store
  - In C this is accomplished by using void* pointers

Our first ADT: Stacks

- A stack is a linear list in which items are added at one end, and removed from the same end
- It is a LIFO structure (last in, first out)
  - Think of it like a stack of books –we can put another book on top, or remove the top one
- There are two main operations we can do on a stack: push a new item on it, and pop the top item off
- We'll to define a C data type called "Stack" so that we can declare variables of this type and manipulate them in various ways

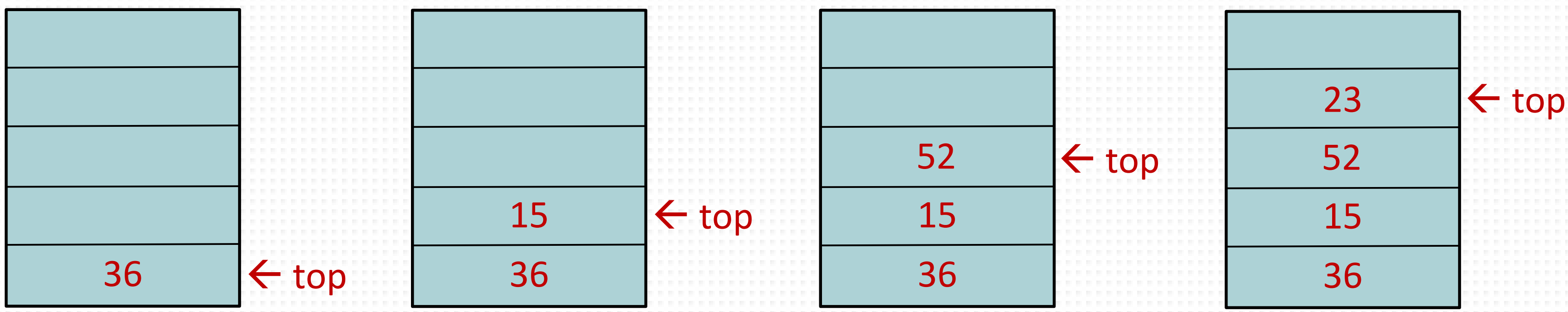
Stacks

- To illustrate stack concepts, let's use a stack of integers
  - But a stack could contain almost any data if we use void* instead
- One use for a stack is reversing a sequence of numbers
  - We might receive the list of numbers 36, 15, 52, 23 and want to change it to 23, 52, 15, 36
- Here's what the stack might look like with values in it:



Stacks

- We could push each item onto our stack as it's received
  - First we'd push 36, then 15, etc.



- When we've received all the data, we could pop each item off the stack
  - Our stack would return 23 first, then 52, 15, 36
- The order of the numbers is reversed

Stack operations

- In addition to push and pop, we'll need
  - A way to initialize a stack
  - A way to check if the stack is empty
- The initialization sets up an empty stack – nothing in it
- The empty check will be useful in loops
- Our stack operations are therefore
  push, pop, initStack and isEmpty

Stack implementation using an array

- Our stack itself is a series of integers
- We'll implement the stack using an array
  - The array provides a place to store the contents of the stack
- We'll make the size of the array equal to MAXSTACK
- To keep track of how many numbers are stored on the stack, we'll use a variable to store which array index is the top of the stack
  - The special index value -1 will mean that the stack is empty
- We'll use a struct to store all the information for the stack

Stack data type

```
typedef struct {
    int top;            // Top of stack
    int ST[MAXSTACK]; // Array containing stack data
} Stack;
```

- The structure shown above represents one stack
- We’ll work with one stack (variable ‘S’ below)

```
static Stack S;  // Variable S has source-file scope
```

Stack overflow

- The valid values for ‘top’ are from 0 to MAXSTACK – 1
- If we get a value greater than or equal to MAXSTACK, this will be considered a stack overflow error
  - In the C array-based implementation it’s actually index out of bounds
- Too many things were pushed on the stack!
  - Stack overflow errors may come up again later in the course when use some recursive algorithms

The initStack function

- Let's write the initStack function
  - It should set 'top' to -1 indicating the stack is empty

```
void initStack(void) {
    S.top = -1;
}
```

- We can call this function like this:

```
initStack();
```

The isEmpty function

- Let's write the isEmpty function
  - It should return true if the stack is empty, false otherwise

```
bool isEmpty(void) {
    return (S.top == -1);
}
```

- The special 'top' value of -1 means the stack is empty

The push function

- To add an integer to the stack we must
  - Add 1 to ‘top’
  - Set array index ‘top’ to the desired integer value
  - We should also check for stack overflow!
  - Remember S.ST is the array, and S.top is the top index

```
void push(int num) {
    if (S.top == MAXSTACK - 1) {
        printf("Stack Overflow\n");  // Stack is full
    } else {
        ++S.top;                     // Update top index
        S.ST[S.top] = num;  // Put 'num' on the stack
    }
}
```

The pop function

- Finally, to remove an integer from the stack we must
  - Get the integer value in array index ‘top’ (we’ll return it)
  - Decrement ‘top’ by 1
  - This time we need to watch out for an empty stack

```
int pop(void) {
    if (isEmpty()) {
        return ROGUEVALUE; // Special value indicates error
    } else {
        int result = S.ST[S.top];  // Get item
        --S.top;                   // Update top index
        return result;             // Return item
    }
}
```

Exercise 1: Stack with integers

- Implement the stack data structure described on the preceding slides
  - Choose reasonable values for MAXSTACK and ROGUEVALUE
- Add a main function to test the stack:
  - Initialize the stack using initStack()
  - Push the numbers 36, 15, 52, 23
  - Pop all the numbers and print them in a loop
    - Stop your loop when the stack is empty
    - Do the numbers come out in the expected order?

Exercise 2: Stack with characters

- Create a module (charStack.c source file and .h header file) containing all code necessary for working with a stack of chars
- Modify the Stack code from Exercise 1 so that it stores an array of chars
- Create a main function in another source file
  - Inside your main function input a string from the user, then push all characters in the string on the stack in order
- Pop all the chars off the stack and print them
  - Use a while loop that continues while the Stack is not empty
  - Note how now the string is reversed

Exercise 3: Binary conversion

- Use the integer stack code from Exercise 1 to write a program that converts decimal numbers to binary
- Follow these pseudocode steps

```
Initialize empty stack S
Read a number, num
while (num > 0)
    push (num % 2) on stack S
    num = num / 2
end while
while (S is not empty)
    print pop(S)
end while
```

Queues

Not so different from stacks…

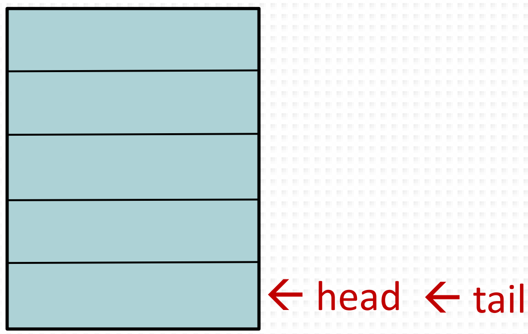
Queues

- A queue is a linear list in which items are added at one end, and removed from the *other* end
- It is a FIFO structure (first in, first out)
  - Imagine a line to get on a ride at an amusement park. The first person to get in line rides first, the second person rides next, and so on… the last person in line rides last.
- We use two index variables to manage a queue
  - Head – indicates which element is at the front of the queue
  - Tail – indicates which element is at the back of the queue
- There are two main operations we can do on a queue: enqueue an item at the back , and dequeue the front item

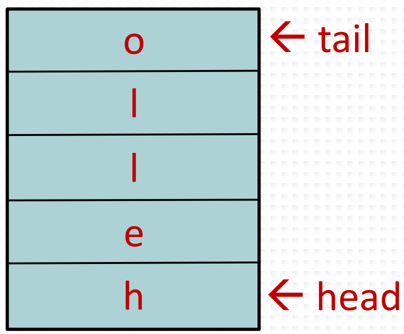
Queues

- To illustrate let's use a queue of characters
  - As with stacks we could use any data type or void* instead
- Queues are often used as data buffers (temporary holders) for input & output, audio or video streaming, networking etc.
- We might receive the sequence of characters h e l l o and put them in a buffer (queue) until the program can process them
- Here's what the queue would look like:

Empty queue

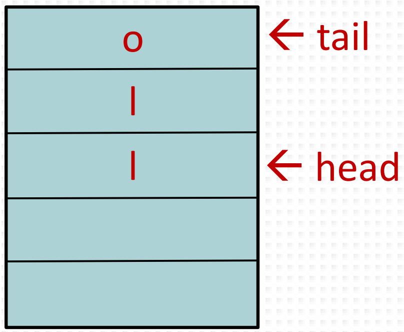


Queue containing "hello"



Queue operations

- When head == tail the queue is empty
- When we enqueue items the tail "pointer" advances
- When we dequeue items , the head "pointer" advances toward the tail pointer (the order of items is not reversed)
- Here's the queue after removing (dequeueing) two chars



Note: The behaviour of head & tail pointers in these pictures is simplified (not precise). For the real behaviour see the code!

Queue operations

- In this queue implementation the head and tail "pointers" are index numbers of an array
  - In our next implementation we'll actually use pointers!
- As before we'll need
  - A way to initialize a queue
  - A way to check if the queue is empty
- Our queue operations are therefore enqueue, dequeue, initQueue and isEmpty
- We'll implement the queue using an array of char
  - The size of the queue array is MAXQ
- We'll use two variables to store the head and tail indexes

Queue data types

- We'll use a struct to store all the information for the queue

```
typedef struct {
    int head;           // Front of queue
    int tail;           // Back of queue
    char QA[MAXQ];      // Array containing queue data
} QType;
```

- The structure shown above represents one queue which we'll use to create a variable like this

```
static QType Q;  // Variable Q has source-file scope
```

The initQueue function

- To initialize a queue, we set the head and tail to zero

```
void initQueue(void) {
    Q.head = Q.tail = 0;
}
```

- Note that the queue can be empty with any valid values for head & tail as long as they are the same

The isEmpty function

- To check if a queue is empty, we just compare the head and tail values

```
bool isEmpty(void) {
    return (Q.head == Q.tail);
}
```

- If the tail has "caught up" with the head, we know the queue has been fully dequeued (it's empty)

Circular queue

- As we add items the tail index increases
- As we remove items the head index increases
- Pretty soon the tail will reach MAXQ (the “end” of the array), however there will probably be empty space available at the “start” of the array
- To reuse the space that’s freed when we dequeue, most queues are implemented as circular queues
  - The head and tail indexes *wrap around* when they reach MAXQ
- In this way a queue can keep going forever, constantly filling and emptying, using a fixed area of memory

Circular queue – empty or full?

- When a circular queue gets full the tail index may reach the head index
  - It can do this because it wraps around
  - Then head == tail
- So if head == tail, does this mean the queue is full or empty?
- We have to be careful so the code doesn't get confused about these two cases!
  - Check when the tail reaches the head as the queue grows and not let it actually happen!

The enqueue function

- To add an item to the queue we must
  - Add 1 to ‘tail’… if ‘tail’ is past the end of the array, set it to zero
  - Set array index ‘tail’ to the desired item value
  - We should also check for queue full condition!
  - Remember Q->QA is the array, Q->head and Q->tail are the indexes

```
void enqueue(char ch) {
    int newTail = Q.tail + 1;  // Calculate new tail index
    if (newTail >= MAXQ)
        newTail = 0;           // Wrap around
    if (newTail == Q.head) {   // Check if queue is full
        printf("Queue is full!");
    } else {
        Q.tail = newTail;
        Q.QA[Q.tail] = ch;     // Put item in the queue
    }
}
```

Why do we need variable ‘newTail’?

The dequeue function

- To remove an item from the queue we must
  - Check if the queue is empty, if so don’t continue
  - Add 1 to ‘head’… if ‘head’ is past the end of the array, set it to zero
  - Return the array element at index ‘head’

```
char dequeue(void) {
    if (isEmpty()) {
        return ROGUEVALUE;     // Special error value
    } else {
        ++Q.head;              // Update head index
        if (Q.head >= MAXQ)
            Q.head = 0;        // Wrap around
        return Q.QA[Q.head];   // Return item
    }
}
```

Exercise 4: Queue of characters

- Implement the queue data structure described on the preceding slides
  - Choose reasonable values for MAXQ and ROGUEVALUE

- Add a main function to test the queue:
  - Create a queue using initQueue()
  - Enqueue the characters of your name
  - Dequeue all the characters and print them on one line using a loop
  - Does it work (do characters appear in the right order)?

Exercise 5: Unbalanced enqueue/dequeue

- Change your program from Exercise 4 so that it prompts for an input string and enqueues the characters typed
- Next it should prompt for the number of characters to dequeue, and dequeue & print that many characters
- Repeat this over and over in a loop
- Try your program… don't dequeue all the characters, see what happens to the characters you leave in the queue
  - Also try to dequeue "too many" characters

Queues in real applications

- Keep in mind that using a queue in these simple example programs isn't very useful
- In real applications queues (buffers) are often used for smooth communication between different independent parts of a program, or between different programs communicating over a network, or to read and write I/O devices. Example: Video/audio streaming