Welcome!

COMP1511 18s1 Programming Fundamentals

Stacks + Queues + ADTs

Andrew Bennett

<andrew.bennett@unsw.edu.au>

Overview

after this lecture, you should be able to ...

have a basic understanding of stacks and queues

have a basic understanding of ADTs

know the difference between **concrete** and **abstract** types

(note: you shouldn't be able to do all of these immediately after watching this lecture. however, this lecture should (hopefully!) give you the foundations you need to develop these skills. remember: programming is

like learning any other language, it takes consistent and regular practice.)

Admin

Don't panic!

assignment 3 out now!

this week's tute/lab help you get started

week 10 weekly test due thursday

don't forget about help sessions! see course website for details

introducing: stacks

Stacks

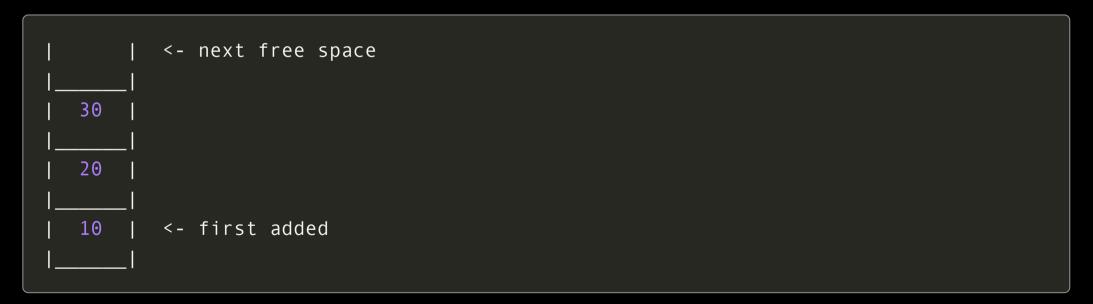
stacks are a type of data structure

(a way of **organising** data)

a **stack** is a collection of items such that the **last** item to enter is the **first** one to exit

"last in, first out" (LIFO)

based on the idea of a stack of books, or plates



Stack

a **stack** is a collection of items such that the **last** item to enter is the **first** one to exit

"last in, first out" (LIFO)

essential stack operations:

...

push() - add new item to stack
pop() - remove top item from stack

additional stack operations:

top() - fetch top item (but don't remove it)
 size() - number of items
 is_empty()

Stack Applications

page-visited history in a web browser

undo sequence in a text editor

checking for balanced brackets

HTML tag matching

postfix (RPN) calculator

chain of function calls in a program

Implementing a Stack

there are several different ways we can implement a stack

(aka actually write the C code to make a stack)

using an array

using a linked list

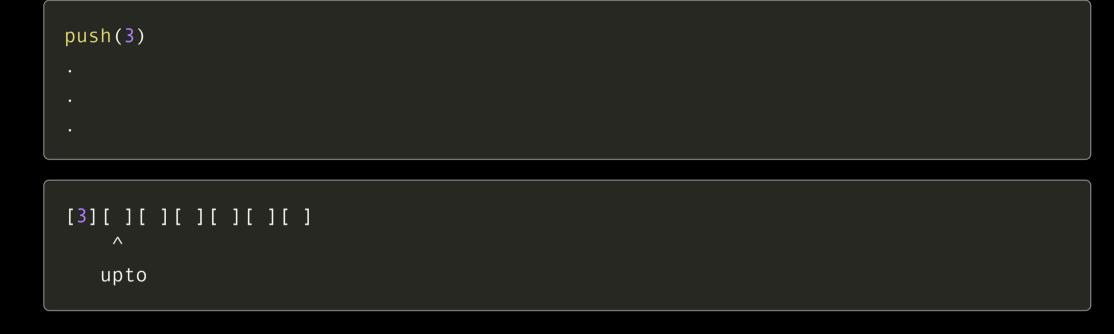
(+ others)

```
struct stack_internals {
    int array[MAX_SIZE]; // holds the values
    int upto; // the index of the next free slot
};
```

```
[ ][ ][ ][ ][ ][ ][ ] // (the array)

^
upto
```

10



11

push(3) push(1)		
[3][1][][][][][] ^		
upto		

12

we can use an **array** to store the stack by keeping track of where we're up to in the array

13

we can use an **array** to store the stack by keeping track of where we're up to in the array

push(3)
push(1)
push(4)
pop() // returns 4

[3][1][][][][][] ^ upto

```
// making a stack
struct stack_internals s = {0}; // initialise to 0
// pushing "5" to the stack
s.array[s.upto] = 5;
s.upto++;
// popping from the stack
s.upto--;
int value = s.array[s.upto];
// value is 5
```

15

a stack can be implemented using a linked list, by adding and removing at the head

struct stack_internals { struct node *head; };

push(3)

(3) -> X

^ head

a stack can be implemented using a linked list, by adding and removing at the head 16

<pre>struct stack_internals { struct node *head; };</pre>
push(3)
push(1)

(1) -> (3) -> X ^ head

a stack can be implemented using a linked list, by adding and removing at the head 17

struct stack_internals {
 struct node *head;
};
push(3)

(4) -> (1) -> (3) -> X ^ head

push(1)

push(4)

a stack can be implemented using a linked list, by adding and removing at the head 18

struct stack_internals {
 struct node *head;

};

push(3)
push(1)
push(4)
pop() // returns 4

(1) -> (3) -> X ^ head

a stack can be implemented using a linked list, by adding and removing at the head 19

```
// making a stack
struct stack_internals s = {0}; // initialise to 0
// pushing "5" to the stack
struct node *node = new_node(5); // make a new node
node->next = s.head; // add before start of list
s.head = node; // update list to start here
// popping from the stack
int value = s->head->data;
struct node *tmp = s->head; // keep track so we can free it
s->head = s->head->next; // update list start
free(tmp);
```

Using a Stack

we can use either of these methods to implement a stack

20

(or another approach!)

I write code to implement a stack, you need to use a stack, so you use my code

but what if the implementation changes?



works... anywhere!



Concrete vs Abstract

struct stack_internals { // ...

};

a type is...

concrete

if a user of that type has knowledge of how it works

a type is...

abstract

if a user has no knowledge of how it works

Concrete vs Abstract

struct stack_internals { // ... };

a concrete type is "right here": if you can see the type, you can use it

Concrete vs Abstract

you cannot **change the insides** of the type without breaking current software 25

we couldn't, for example, easily **switch between** stack implementations (array vs list)

Abstraction

our old friend, abstraction

26

use functions to interact with the stack, **push pop** etc

doesn't really matter how the **implementation** works... only that the **interface** is correct.

Hiding Structures

typedef struct stack_internals *stack;

we can now refer to **stack**, without knowing what's in **struct stack_internals**...

we cannot dereference (stab) it but it can move around the system as an opaque value.



separating the implementation from the interface



implementing a stack ADT

Why a Stack ADT?

if we implement our stack as an ADT

30

we can change the implementation

without affecting how to use the stack

Stack - Abstract Data Type - C Interface

31

// use `stack` to refer to a pointer to the stack struct
typedef struct stack_internals *stack;

// pass the pointer into the stack functions
// (rather than trying to modify the struct directly)
stack stack_create(void);
void stack_free(stack stack);
void stack_push(stack stack, int item);
int stack_pop(stack stack);
int stack_is_empty(stack stack);
int stack_top(stack stack);
int stack_size(stack stack);

Stack - Abstract Data Type - using C Interface

we can only interact with the stack using its **interface** functions 32

C	+	С	\boldsymbol{c}	2	S	
С	L	а	C	N	2	,

s = stack_create();

stack push(s, 10);

stack push(s, 11);

stack push(s, 12);

printf("%d\n", stack_size(s)); // prints 3

printf("%d\n", stack top(s)); // prints 12

printf("%d\n", stack pop(s)); // prints 12

printf("%d\n", stack pop(s)); // prints 11

printf("%d\n", stack_pop(s)); // prints 10

Stack - Abstract Data Type - using C Interface

we can only interact with the stack using its **interface** functions 33

we can't **dereference** the pointer or access the struct fields

```
stack s = stack_create();
// note: if we tried to do this,
// we would get a compile error
// we can't see inside the struct, how do we know
// if it has an `array` field?
s->array[0] = 10;
// how do we know if it has a `size` field?
printf("%d", s->size);
```

Stack - Abstract Data Type - using C Interface

34

implementation of stack is **opaque** (hidden from user); user programs can not depend on how stack is implementated.

stack implementation can change *without* risk of breaking user programs.

information hiding is crucial to managing complexity in large software systems.

Stack - Abstract Data Type - switching implementations

we can easily change which implementation we use

// inside stack_user.c
stack s = stack_create();
stack_push(s, 5);
stack_push(s, 10);
printf("%d, stack_pop(s));
printf("%d, stack_pop(s));

\$ dcc -o stack stack_user.c stack_list.c
\$./stack
10
5

\$ dcc -o stack stack_user.c stack_array.c
\$./stack
10
5

