Cosc 241
Programming and Problem Solving
Lecture 14 (11/4/2019)
Building a stack

Lech Szymanski
lechszym@cs.otago.ac.nz

Keywords: stack, amortized analysis, exceptions

Stack ADT

```
public interface Stack241<T> {

  public void push (T element);
  public T pop();
  public T peek();
  public boolean isEmpty();
  public int size();

}
```

Array implementation

- Use an array to keep track of the stack contents.
- A variable `size` will track the size of the stack.
- Problems?
  - Generic array creation is not possible (why not?)
  - How do we cope if “too much” gets pushed onto the stack?

Performance analysis

- What are the time and space costs of each of the operations?
- Space first – we allocate $O(1)$ space initially (because of the default capacity).
- Later, when we expand capacity we never have more than twice as much available as needed, so that's $O(n)$ where n is the maximum number of items we ever include in the stack.
- Time for `pop`, `peek`, `isEmpty` are all clearly $O(1)$.
- The push operation is also $O(1)$, except when we need to expand the stack – what then?

Amortized cost

- It seems unfair to associate the cost of the stack expansion to the single item that caused it,
- To be completely accurate we *do* have to say that a `push` operation can be $O(n)$.
- But practically, we can think of the cost of expansion as being spread over all the elements present when it happened.
- Since there are n elements present, and the expansion is $O(n)$, that means that in an amortized sense the `push` operation is still $O(1)$.
- Essentially, we imagine borrowing a little bit of time every time we do a push, and spending it all when the expansion is needed.
- Operationally though, we may observe occasional slow downs if using very large stacks.

Exceptions etc.

- The `peek` and `pop` operations need to do *something* on empty stacks.
- One “solution” is to have them return `null` in these cases – in some contexts that’s actually not bad.
- More in keeping with the spirit of the ADT is that such operations should generate exceptions, and the user should be responsible for throwing same, or enclosing in a try-catch block.
- General principle – each ADT tends to come with its own subclasses of `Exception` to describe the exceptions that it might generate.