CS180 - Stacks & queues

Announcements

- HW 2 is out, due in 1 week
- Midterm 1 in 2-3 weeks (?)
  (more next time on date)
Last time: Stacks

Operations:
- push
- pop

push(5)
push(9)
push(0)
pop() → 0
pop() → 9
push(2)
int num(5);

We coded the main functions.

What is left?

Hint: Consider this command

\[
\text{Stack 1} = \text{Stack 2}
\]

As default, C++ will run \( \neq \) for each private piece of data.

Problems

1. Shallow copy

2. Memory leaks - need a destructor

Shallow copy.

Stack 2

\[
\text{t = 5, capacity = 7}
\]

Stack 1

\[
\text{t = 5, capacity = 7}
\]
Operator = 

```
ArrayStack & operator=(const ArrayStack & other) {
  if (this != &other) {
    capacity = other.capacity;
    t = other.t;
    delete [] S;
    S = new Object[capacity];
    for (int i = 0; i <= t; i++)
      S[i] = other.S[i];
  }
  return *this;
}
```
Destructor: Is called when function ends (or scope ends).

```
~Array Stack() {
    delete [] S;
    3
}
```

Aside:
```
while (...) {
    int num;
    ...
}
```

C++ automatically deletes private variables — just not what variables point to. Be careful!
Copy constructor

ArrayStack (const ArrayStack& other):
  capacity (other.capacity), t(other.t)
  
S = new Object[capacity];
for (int i = 0; i < t; i++)
  S[i] = other.S[i];

in main:
  ArrayStack <int> stack2(stack1);
Final Code

on webpage

Note: Different implementations are possible!

See book versus webpage versus lecture version.
Running Times:

<table>
<thead>
<tr>
<th>Function</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>size is Empty</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>top</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>push</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>pop</td>
<td>( O(1) )</td>
</tr>
</tbody>
</table>

Size usage?

- \( \text{not } O(n) \)
- \( O(\text{capacity}) \)

\( \leftarrow \text{return } S[e..j] \)

Copy Constructor: \( O(n) \) (for loop)
Sec. 4.2.3 - Function calls + stacks

C++ actually keeps a private stack, called the run-time stack, to keep track of local variables.

Why is this data structure ideal?

```c
    function {
        while {
            if { ... } ...
        }
    }
```

(see p. 166-168 for more detail.)
Queues: Another way of storing a list
First in, first out (FIFO)

Two main functions:

enqueue(0): Insert object 0 at the
    rear of the queue
    push

dequeue(): Remove & return the object
    at the front of the queue
    pop
Other operations

- `size()`
- `is Empty()`
- `first()` *(same idea as top)* 
  does not remove just returns first element
<table>
<thead>
<tr>
<th>Operation</th>
<th>Output</th>
<th>Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>enq(5)</td>
<td>-</td>
<td>[6]</td>
</tr>
<tr>
<td>enq(3)</td>
<td>-</td>
<td>[5, 3]</td>
</tr>
<tr>
<td>dq()</td>
<td>5</td>
<td>[3]</td>
</tr>
<tr>
<td>enq(7)</td>
<td>-</td>
<td>[3, 7]</td>
</tr>
<tr>
<td>dq()</td>
<td>3</td>
<td>[7]</td>
</tr>
<tr>
<td>Fnt()</td>
<td>7</td>
<td>[7]</td>
</tr>
<tr>
<td>dq()</td>
<td>7</td>
<td>[ ]</td>
</tr>
<tr>
<td>dq()</td>
<td>error</td>
<td>[ ]</td>
</tr>
<tr>
<td>is Empty()</td>
<td>True</td>
<td>[ ]</td>
</tr>
<tr>
<td>enq(9)</td>
<td>-</td>
<td>[9]</td>
</tr>
<tr>
<td>enq(7)</td>
<td>-</td>
<td>[9, 7]</td>
</tr>
<tr>
<td>size()</td>
<td>2</td>
<td>[ ]</td>
</tr>
<tr>
<td>enq(3)</td>
<td>-</td>
<td>[9, 7, 3]</td>
</tr>
<tr>
<td>enq(5)</td>
<td>-</td>
<td>[9, 7, 3, 5]</td>
</tr>
<tr>
<td>dq()</td>
<td>9</td>
<td>[7, 3, 5]</td>
</tr>
</tbody>
</table>
Alright - let's think about the setup:

template <typename Object>
class Queue {
    public:
        int size() const;
        bool isEmpty() const;
        const Object& front() const;
        void enqueue(Object obj);
        Object dequeue();
}