Data Structures and Objects
CSIS 3700

Lab 6 — Array Doubling

Goal

Modify the templated Stack and Queue classes to use arrays that grow when full.

Motivation

Our first versions of the Stack and Queue classes used arrays to hold the data. This has the advantages of speed and simplicity; however, because an array is used, the containers have a fixed size.

We could use a linked structure instead. As with an array-based implementation, all of the stack and queue operations still have $O(1)$ run-time. Unlike arrays, linked structures can grow to accommodate additional data. However, in practice, linked structures are slower than arrays. Having to call memory management functions for each addition and removal slows down access.

Array doubling provides a middle ground. Arrays are used for faster access and memory management functions are called when more space is needed. The trick is to limit the number of times memory managment is needed.

How It Works

Consider our current Stack template. Note that it declares an array for holding data:

```
49  private:
50   StackType
51     data[STACK_SIZE];
52   int
53     count;
```

Since both arrays and pointers can be subscripted, we could replace the array with a pointer:

```
49  private:
50   StackType
51     *data;
52   int
53     count;
```

Note that one of the big differences between arrays and pointers is that pointers do not get space to be used like an array; we have to provide that separately. To do this, we'll need to allocate the space in the constructor, before we start adding data. We can deallocate the space in the destructor.

```
13  public:
14   Stack() {
15     data = new StackType[STACK_SIZE];
16     count = 0;
17   }
18   ~Stack() { delete[] data; }
```

Now, the Stack class works just as it did initially, including the possibility of filling and throwing exceptions. However, the current approach gives us another choice of action when the array fills — we can create a larger array.

To begin, we need to keep track of how large the current array is. Initially, this is just the `STACK_SIZE` constant; however, over time, the array may increase in size multiple times, so we'll add a new variable to keep track of the current size.

```
66  private:
67   StackType
68    *data;
69   int
70    count,
71    capacity;
```

The capacity must be initialized in the constructor:

```
10  public:
11   Stack() {
12     data = new StackType[STACK_SIZE];
13     capacity = STACK_SIZE;
14     count = 0;
15   }
```

The only other change that needs to be made is in handling a push onto a full stack; all other actions are unchanged, one of the advantages of this technique. To increase space, do the following:

1. Begin by checking to see of `count` equals `capacity` instead of `STACK_SIZE`
2. Calculate a new capacity that is twice as large as the current capacity.
3. Dynamically allocate a new array with the new capacity as the number of elements.
4. Copy all elements from the current array to the front of the new array.
5. Deallocate the current array and point to the new array.
6. Remember the new array and capacity.

If for some reason the memory allocation fails, then an exception is thrown.

The resulting code looks like this:

```
22 void push(const StackType &d) {
23
24   // out of space? make more space!
25   // NOTE: STACK_SIZE is now capacity
26   if (count == capacity) {
27     int
28       tmpCap = 2 * capacity;
29     StackType
30       *tmpData = new StackType[tmpCap];
31
32     // this shouldn't happen in practice
33     if (tmpData == nullptr)
34       throw std::overflow_error("Stack_is_full");
35
36     // copy data from old array into new array
37     for (int i=0;i<capacity;i++)
38       tmpData[i] = data[i];
39
40     // toss the old array
41     delete[] data;
42
43     // remember the new array and its capacity
44     data = tmpData;
45     capacity = tmpCap;
46   }
47
48   // and we continue on like nothing happened
49   data[count] = d;
50
51   count++;
52 }
```

Analysis

To analyze the cost for stack operations using array doubling, note first that the only operation that has changed in any significant way is the push operation; other operations are either unchanged or changed in a way that does not add significant time.

The following table shows the costs involved with a certain number of pushes, assuming that the initial capacity is k.

Pushes	Push Cost	Cost From Copy	Total Cost	Average Cost	New Capacity
1	1	0	1	1	k
2	1	0	2	1	k
⋮	⋮	⋮	⋮	⋮	⋮
k	1	0	k	1	k
$k+1$	$k+1$	k	$2k+1$	$\frac{2k+1}{k+1} < 2$	$2k$
$k+2$	1	0	$2k+2$	$\frac{2k+2}{k+2} < 2$	$2k$
⋮	⋮	⋮	⋮	⋮	⋮
$2k$	1	0	$3k$	$\frac{3}{2}$	$2k$
$2k+1$	$2k+1$	$k+2k=3k$	$5k+1$	$\frac{5k+1}{2k+1} < 2.5$	$4k$
$2k+2$	1	0	$5k+2$	$\frac{5k+2}{2k+2} < 2.5$	$4k$
⋮	⋮	⋮	⋮	⋮	⋮
$4k$	1	0	$7k$	$\frac{7}{4}$	$4k$
$4k+1$	$4k+1$	$k+2k+4k=7k$	$11k+1$	$\frac{11k+1}{4k+1} < 2.75$	$8k$

The table implies, and it can be shown, that the average cost for each push that increases the size approaches 3 in the limit and always remains below 3 for all pushes. Thus, the average cost per push remains in $O(1)$ as it does normally.

Queues and Array Doubling

Queues can also use array doubling for increased storage capacity. It works in much the same way as it does with stacks; the main difference is in the copying of data into the new array. In copying a stack with capacity k, the data is known to be stored starting with position 0 and filling all slots. However, the start of a queue — its head — can be at any position, with data wrapping around at the end of the array. If we just do a position-by-position copy, then we will need to wrap around to the beginning when the head reaches the middle of the new array, but won't until it reaches the end. The solution is to copy the head position into the new array's position 0, the next element goes into position 1, and so on, so that the data does not wrap. Then, after copying, reset the head index to its starting position and adjust the tail index to be the first unused position in the new array — this will be equal to the current capacity.

To do this, make the same changes to the template queue.h that you made to the stack.h file. Then, change the for loop to read like this:

```
1       for (int i=0;i<capacity;i++)
2         tmpData[i] = data[(head+i+1)%capacity];
3
4       head = tmpCap - 1;
5       tail = capacity;
```

And that's it, the queue should now be extendable.

▸*Warning!*

`QUEUE_SIZE` should only be used in the constructor, to set `capacity` and to allocate the initial array. Use `capacity` everywhere else when checking for a full queue or for wrapping `head` and `tail`.

Advanced Material

Note: You don't have to do these for the lab; I'm only including them for completeness.

▹Altering the rate of increase

It isn't necessary to double the size of the array each time more space is needed. Any multiplier greater than one will work. The closer to one the multiplier becomes, the larger the limit becomes on the average per push, but it will remain a constant. The larger the multiplier becomes, the more space you are likely to waste. You can also add a constant amount of space instead of multiplying; however, the average push cost is no longer a constant in the limit.

In my personal implementation of a template stack, I add four default parameters to the constructor: an initial capacity, a numerator and denominator that specify the capacity multiplier as a fraction, and an amount added to the capacity in addition to the multiplier. This allows maximum flexibility, as I can opt for a quick increase in space, a slow increase, a constant increase or no increase at all. When the code calculates a new capacity, it checks that value against the current capacity; if the new capacity is not larger then it throws an exception and does not reallocate space.

▹Shrinking the array

It is also possible to reduce the size of the array. If you double the size when the array fills, then you can cut the size in half when the count is reduced to 25% of capacity. The cost of doing so does not change the $O(1)$ behavior of the pop operation.

In practice, I don't do this. I don't find it to be beneficial, as places where I use array doubling typically have space to spare, and in space-restricted environments I'm not likely to use an array doubling implementation. This doesn't make it wrong, just not something I personally use.

What to turn in

Please turn in your modified stack.h and queue.h files. Please submit them in a single, compressed file.