Stacks and Queues

Notes. This document was created by Sasha Avreline in the summer of 2020 as an extra study aid to BCS students taking CPSC 221. All code can be found at

`https://github.com/savreline/tutorials/tree/master/cpp/stacks`

The problem considered in sections 2-4 is LeetCode problem 232.
The problem considered in section 5 is LeetCode problem 225.

Contents

1 Summary 2

2 Implementing a Queue using Two Stacks:
$\mathcal{O}(n)$ Iterative version 4

3 Implementing a Queue using One Stack:
$\mathcal{O}(n)$ Recursive version 5

4 Implementing a Queue using Two Stacks:
$\mathcal{O}(1)$ Iterative version 7

5 Implementing a Stack using a Queue 8

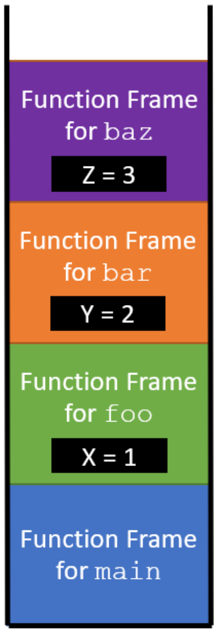
1 Summary

- **Stack:** LIFO (last in, first out); like a stack of paper
- **Queue:** FIFO (first in, first out); like a queue at a store
- All pushes and pops are $\Theta(1)$.
- Required storage is $\Theta(n)$.
- Locating a specific item is $\mathcal{O}(n)$.
- A queue is used to implement breadth-first search (BFS) in a tree or a graph. In a tree, using a queue results in a level-order tree traversal.
- A stack is used to implement depth-first search (DFS) in a tree or a graph. Recursion is closely associated with stacks as the data associated with recursive function calls is stored on the stack portion of memory.

Relationship between Stacks, Function Calls and Recursion. Would function calls behave more like a queue or a stack? Function calls behave like a stack! The last function call is the first one to return, so we have a LIFO situation here.

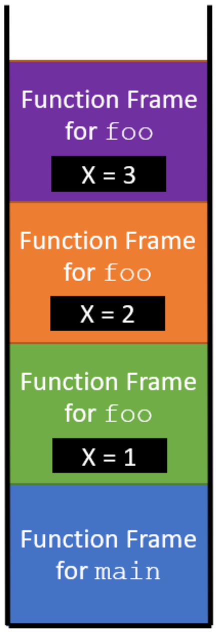
Consider the following code. We have main call foo, foo call bar, bar call baz. We would expect baz to finish its work before returning back to bar and then bar to finish before returning to foo, etc. Whenever a new function call is made, a frame associated with that function call is placed on the stack portion of memory. That frame, among other things, holds the local variables associated with that function call. As functions start to return, their frames are deleted off the stack. So this is why memory uses a stack, so that data associated with functions which were called last and return first could be deleted first!

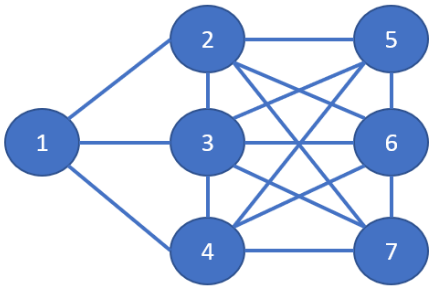
```
void baz() {
    int z = 3;
}

void bar() {
    int y = 2;
    baz();
}

void foo() {
    int x = 1;
    bar();
}

int main() {
    foo();
}
```



What about in the case of recursion? Well we end up placing the frame associated with the same function on the stack several times, once per each recursive call, yet of course with different instances of local variables. So once all of the recursive calls fully "open up", in memory there is a stack that holds instances of all variables that we passed to the function calls and we can make use of those variables as the recursive calls start to return.

```
void foo(int x) {
    x += 1;
    foo(x);
}

int main() {
    foo();
}
```





Relationship between Stacks, Queues, BFS and DFS. Now, consider the graph above. Suppose we wish to traverse the graph starting from node 1. When we visit a node, we wish to use some structure to remember that node's neighbors, so that we can visit them next. We can use either a stack or a queue to do this (it is essentially a debate of whether should use a FIFO or LIFO data structure to store the neighbors).

If we use a queue, then we will

1. Visit node 1 and add (2, 3, 4) to the queue as neighbors of node 1.
2. Dequeue node 2 from the queue to visit next, add its neighbors (5, 6, 7) to the queue.
3. Dequeue node 3 from the queue to visit next, so at this point we are just scanning the next "layer" of nodes that follows node 1 and are doing a *breadth-first search*. We will finish scanning this level before moving onto nodes (5, 6, 7) in the next layer.

If we use a stack, then we will

1. Visit node 1 and add (2, 3, 4) to the stack as neighbors of node 1.
2. Pop node 4 from the stack to visit next, add its neighbors (5, 6, 7) to the stack.
3. Pop node 7 from the stack to visit next, so at this point we are taking the most recent nodes off the stack, going deeper into the graph and are doing a *depth-first search*.

If one were to do a *recursive* tree/graph traversal, one would get a *depth-first search* as stacks and recursion are related. If one were to do an *iterative* tree/graph traversal, one would get a *breadth-first search*, as queues mimic for loops, where items are visited in order.

2 Implementing a Queue using Two Stacks: $\mathcal{O}(n)$ Iterative version

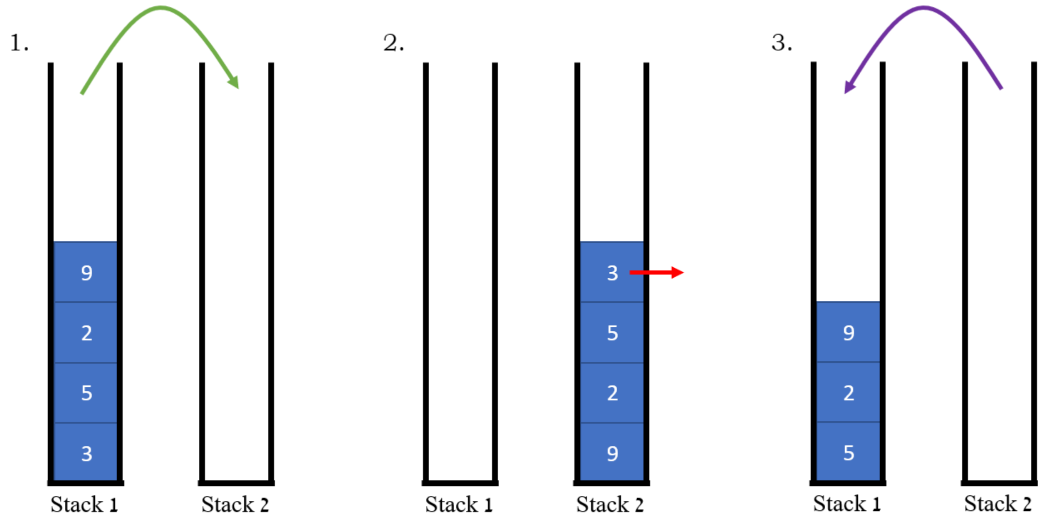
How can we use two stacks to implement a queue? The constructor and the push methods are quite straight forward: we just initialize the two stacks and then upon a push, just push the elements into the first stack.

```
Queue() {
    s1 = stack<int>();
    s2 = stack<int>();
}
```

```
void push(int x) {
    s1.push(x);
}
```

The pop method is where real work needs to be done. We want the bottom element but just have access to the top! In the most basic implementation we would just offload the entire first stack onto the other stack to reach the element at the bottom (in green), grab the bottom element (in red), and place remaining elements back onto the first stack (purple).

```
int pop() {
   while (!s1.empty()) {   // STEP 1
      s2.push(s1.top());
      s1.pop();
   }

   int result = s2.top(); // STEP 2
   s2.pop();

   while (!s2.empty()) {  // STEP 3
      s1.push(s2.top());
      s2.pop();
   }

   return result;
}
```



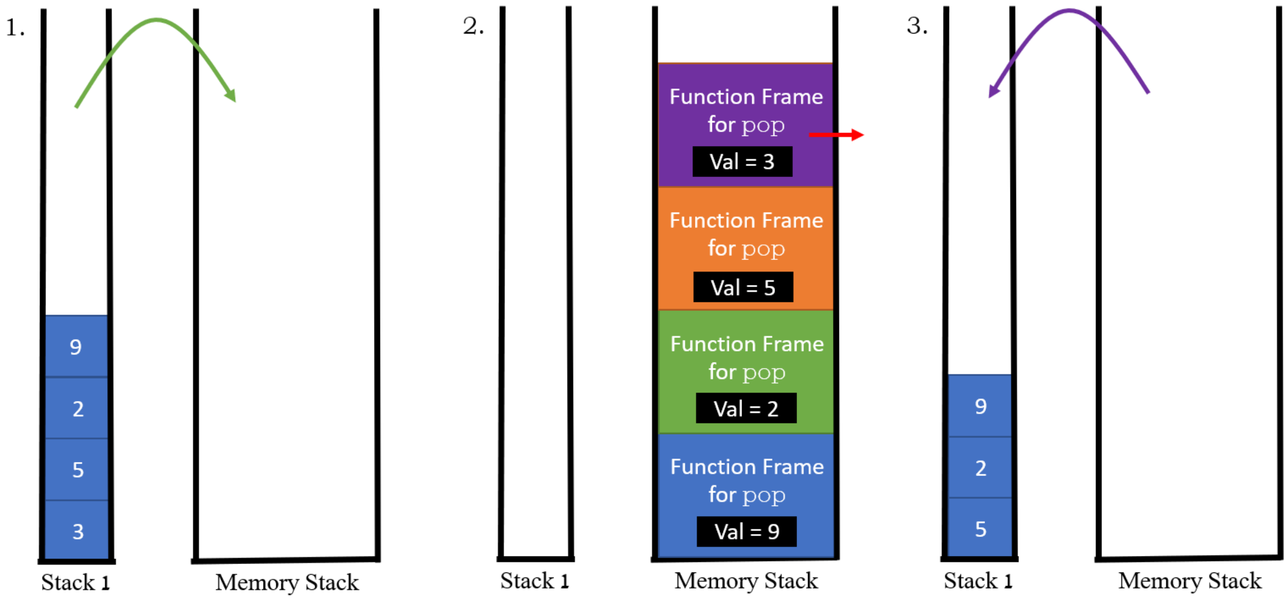
This pop method of course runs in $\mathcal{O}(n)$. More precisely, its runtime is $2n$ as we offload all n elements and then load them back up. It seems a bit wasteful to load all of the elements back onto the original stack once we popped the bottom element. We will address this issue in section 4, but next lets try a recursive version.

3 Implementing a Queue using One Stack: $\mathcal{O}(n)$ Recursive version

Here is the recursive version. The colours used are meant to roughly match pieces of the code from the previous version. In the green code, we do the work that happens *prior* to the recursive calls fully "opening up". We pop the top element of the stack and save it into `val` that is then placed on the memory stack as a local variable that is associated with the specific recursive call.

We reach the base case (in red) once all of the recursive calls have fully "opened up". There we grab the value sitting at the top of the memory stack and will pass it back through the return statements of the recursive calls to the top level call.

```
int pop() {
   int val = s1.top();
   s1.pop();

   if (s1.empty())
     return val;
   else {
     int result = pop();
     s1.push(val);
     return result;
   }
}
```



Once we are done with the base case we start wrapping up the recursive calls and as they return, we run the code in purple that re-assembles the initial stack. It is very critical that this purple line of code is placed *after* the recursive call, as we want to ***delay*** this work of re-assembling the original stack to after all of the calls have fully "opened up" and we managed to save the bottom of the stack into `result`.

Note that in the iterative approach we needed to use *two* stacks and here we are just using *a single* stack. Why? Well, of course, it is because the memory stack acts as the second stack in this case!

Also note the similarities between a while loop and a recursive call, the following two snippets of code are equivalent. This equivalency was kept in mind when translating the green portions of the code between the two versions of implementation.

```
int x = 5;
while (x > 0) {
    x--;
}
```

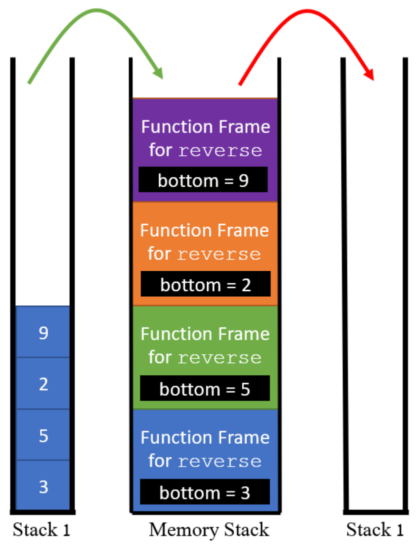
```
void foo(int x) {
    if (x <= 0) return;
    x -= 1;
    foo(x);
}
foo(5);
```

Now, for fun, we could go one step further and write a *double-recursive* function that reverses the stack in place, without using any additional data structures. Well of course it heavily uses the memory stack and has a terrible $\mathcal{O}(n^2)$ runtime, but it reverses the stack in-place!

Here we just make a call to the `pop` function that we know returns the bottom element, and then make recursive calls to `reverse`. Below in purple we have the base case, in green the work done *prior* to the recursive calls, in cyan the recursive call itself and in red the work done *after* all recursive calls have "opened up" and are coming back.

```
void reverse() {
    if (s.empty())
      return;
    int bottom = pop();
    reverse();
    s1.push(bottom);
}
```

Once all recursive calls have "opened up", the memory stack would have the exact copy of the stack we were working with, and this copy would be copied in the reverse order into our stack as the recursive calls are coming back.


Function Frame for reverse
bottom = 9
Function Frame for reverse
bottom = 2
Function Frame for reverse
bottom = 5
Function Frame for reverse
bottom = 3
9
2
5
3
Stack 1
Memory Stack
Stack 1

One should be careful with heavy use of the memory stack: the space there is limited and eventually one would get a "stack overflow error". Imagine if out stack was 200 elements long - that would be hundreds of function frames on the memory stack!

4 Implementing a Queue using Two Stacks: $\mathcal{O}(1)$ Iterative version

Next, let's address the issue of the iterative implementation being inefficient (this section refers back to the code given in section 2). During the pop, after we transferred our original stack into the second stack, saved and popped the bottom element, we then transferred the remaining elements back to the original stack. As mentioned, it seems a bit wasteful to transfer all of the elements back. As in, if pop was called again right away, had we not transferred all the elements back in, we could have done pop in $\Theta(1)$ time!

So, let's keep all of the elements in the second stack and just keep popping from there whenever pop is called. While that is happening, we just got to be sure that we don't push any other elements into that second stack, or the order of elements would be lost. So, we keep pushing the new elements that we are given into the first stack and then transfer elements over into the second stack only when the *second stack is empty* - i.e. enough pops happened to fully empty out that stack. That way the order will be preserved!

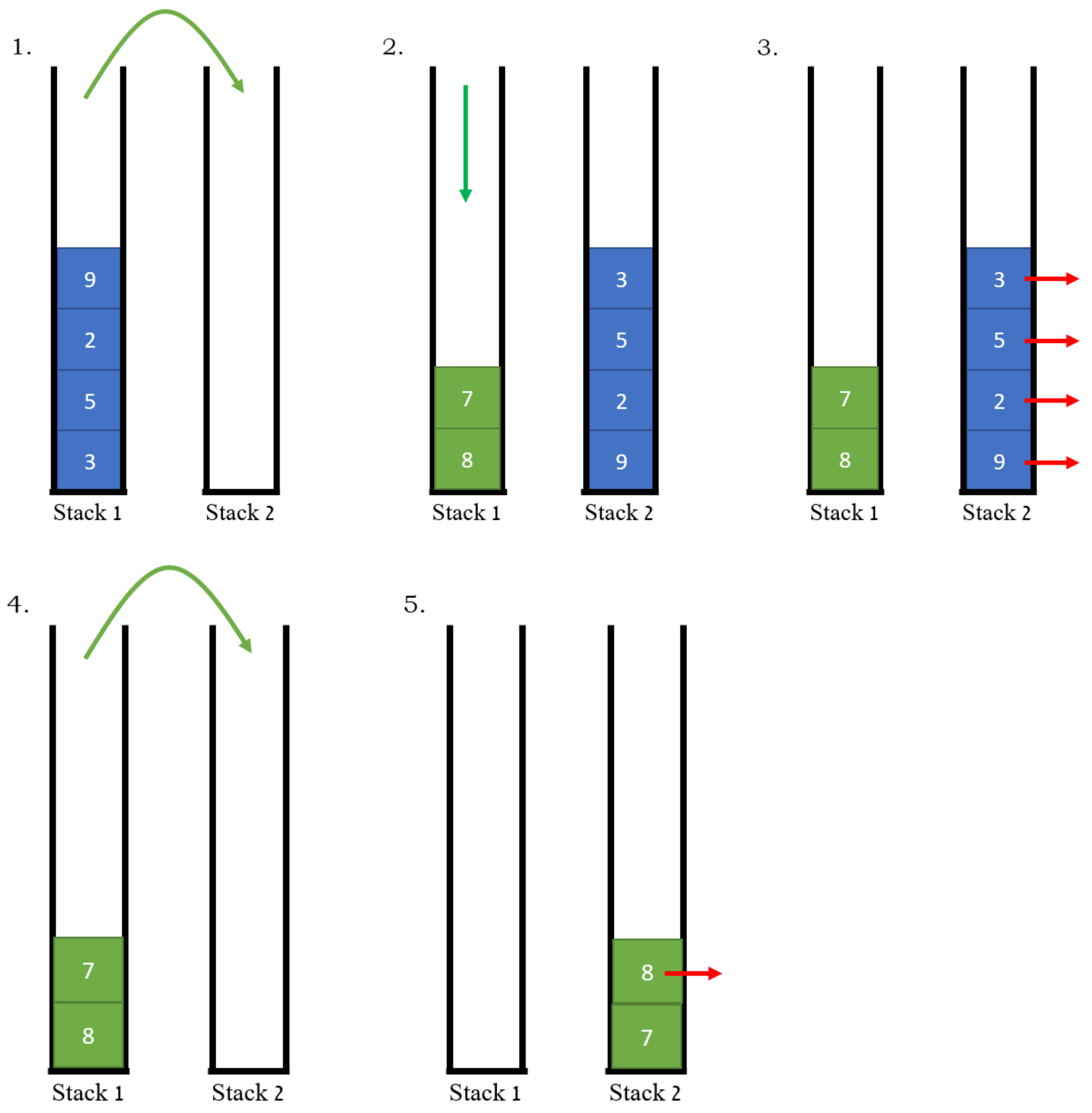
Here is the code, diagram is on the following page. Colour coding here is consistent with section 2:

- The green portion still transfers stack 1 into stack 2 as before. This time; however, we need to wrap this portion into an if statement, as we only want the transfer to happen when stack 2 is completely empty.
- The red portion still pops the item.
- The purple portion is now gone - we *never* transfer anything back to stack 1!

```
int pop() {
   if (s2.empty()) {
     while (!s1.empty()) {
       s2.push(s1.top());
       s1.pop();
     }
   }

   int result = s2.top();
   s2.pop();
   return result;
}
```

The diagrams on the following page show what happens when we add elements 3, 5, 2, 9; then pop 3, and add elements 8 and 7 before doing any more pops. Then by step 4 in those diagrams we have popped all of 5, 2, 9 and are ready to pop 8, which leads to transfer the next batch of elements from stack 1 to stack 2.

The runtime of this method is $\mathcal{O}(1)$ as the odd times when we do have to transfer elements to the second stack averages out with all of the simple pops we could do by just popping the top element off the second stack (this can be shown mathematically).



5 Implementing a Stack using a Queue

Finally let's look at the opposite problem: what if we wanted to implement a stack using a queue? Here things are actually a bit simpler, and that is because we sort of have access to both sides of the queue. When working with the stack, we only had access to the top of the stack, so we had to offload the stack somewhere else to get to the bottom. In the case of the queue, we can pop from the front and push into the back, so we can *rotate the queue in place.*

Upon each push, we push the element in and then run a while loop to rotate the entire queue so that this last element pushed in happens to be at the front of the queue, ready to be popped off. Then in the pop method, we just pop this element off.

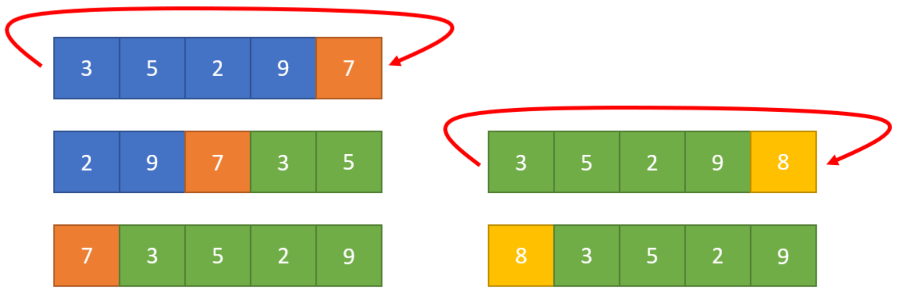
In the following code, in purple we push the element in, in green we rotate the queue and in red we pop the element off. Since when we are rotating the queue, we can't run a `while(q1.empty())` loop as the queue would never be empty. Luckily we have the `size()` method and can save the size into a variable that we would decrement. We don't want to decrement the variable to zero, as that would fully rotate the queue, which is not what we want (we don't want to rotate the last element, that is the element we want to bring to the front).

This method of course runs in $\mathcal{O}(n)$, moreover, in this problem, we can't even use the approach of section 4 to come up with an algorithm that runs in $\mathcal{O}(1)$. This is because even if we had an another queue, moving elements there would just keep them in the same order, unlike in the case of a stack.

Finally, note that we should really do the rotation in the push method. Otherwise if the stack were to have both the pop and top methods, then the top method would rotate the queue without removing the element. If we call the pop method next, it would rotate the queue again, removing the wrong element.

```
void push(int x) {
   q1.push(x);

   int size = q1.size();
   while (size > 1) {
     q1.push(q1.front());
     q1.pop();
     size--;
   }
}

int pop() {
   int result = q1.front();
   q1.pop();
   return result;
}
```

As an example, suppose that we pushed 3, 5, 2, 9, 7 into the queue in that order. Then we immediately rotate the queue so that 7 is at the front (first column of diagrams). Next, if we pop 7 and push in 8, we immediately rotate the queue again, so that 8 is at the front right away (second column of diagrams), ready to be popped.


3 5 2 9 7
2 9 7 3 5
7 3 5 2 9
3 5 2 9 8
8 3 5 2 9