How to implement a queue using two stacks?

Suppose we have two stacks and no other temporary variable.

Is to possible to "construct" a queue data structure using only the two stacks?

algorithm data-structures stack queue

edited Aug 23 '16 at 0:59
Levent Divilioglu
3,896 ● 1 ● 23 ● 66

asked Sep 16 '08 at 3:37
Nitin
4,531 ● 8 ● 18 ● 14

15 Answers

Keep 2 stacks, let's call them `inbox` and `outbox`.

Enqueue:

- Push the new element onto `inbox`

Dequeue:

- If `outbox` is empty, refill it by popping each element from `inbox` and pushing it onto `outbox`
- Pop and return the top element from `outbox`

Using this method, each element will be in each stack exactly once - meaning each element will be pushed twice and popped twice, giving amortized constant time operations.

Here's an implementation in Java:

```java
public class Queue<E>
{

    private Stack<E> inbox = new Stack<E>();
    private Stack<E> outbox = new Stack<E>();

    public void queue(E item) {
        inbox.push(item);
    }

    public E dequeue() {
        if (outbox.isEmpty()) {
            while (!inbox.isEmpty()) {
               outbox.push(inbox.pop());
            }
        }
        return outbox.pop();
    }

}
```

edited Dec 22 '16 at 2:36
rimsky
595 ● 6 ● 22

answered Sep 16 '08 at 4:42
Dave L.
31.2k ● 7 ● 46 ● 54

17 Excellent point! I missed that little detail when I first read Brian's solution. There really isn't any reason to copy the outbox back to the inbox anyway, so that could be why I completely missed that "implementation" detail. :-) – Daniel Spiewak Sep 16 '08 at 4:46

1 Yeah, I almost skipped over that too! If I had edit privileges, I would have just fixed his answer, but this makes it clear. – Dave L. Sep 16 '08 at 4:50

7 The worst-case time complexity is still O(n). I persist in saying this because I hope no students out there (this sounds like a homework/educational question) think this is an acceptable way to implement a queue. – Tyler Sep 16 '08 at 12:56

14 It is true that the worst-case time for a single pop operation is O(n) (where n is the current size of the

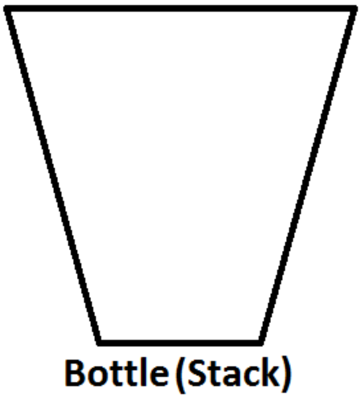
queue). However, the worst-case time for a sequence of n queue operations is also O(n), giving us the amortized constant time. I wouldn't implement a queue this way, but it's not that bad. – Dave L. Sep 16 '08 at 14:06

13 @Newtang a) queue 1,2,3 => **Inbox[3,2,1]/Outbox[]**. b) dequeue. outbox is empty, so refill => **Inbox[]/Outbox[1,2,3]**. Pop from outbox, return 1 => **Inbox[]/Outbox[2,3]**. c) queue 4,5 => **Inbox[5,4]/Outbox[2,3]**. d) dequeue. outbox is not empty, so pop from outbox, return 2 => **Inbox[5,4]/Outbox[3]**. Does that make more sense? – Dave L. Feb 25 '13 at 14:28

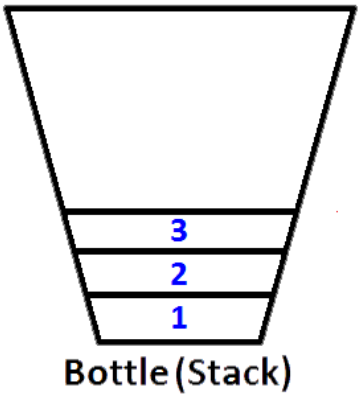
A - How To Reverse A Stack

To understand how to construct a queue using two stacks, you should understand how to reverse a stack crystal clear. Remember how stack works, it is very similar to the dish stack on your kitchen. The last washed dish will be on the top of the clean stack, which is called as **L**ast **I**n **F**irst **O**ut (LIFO) in computer science.

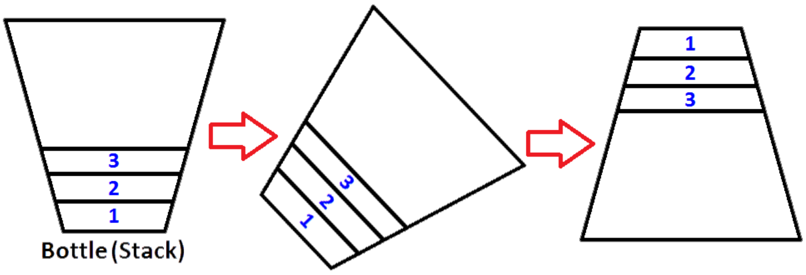
Lets imagine our stack like a bottle as below;



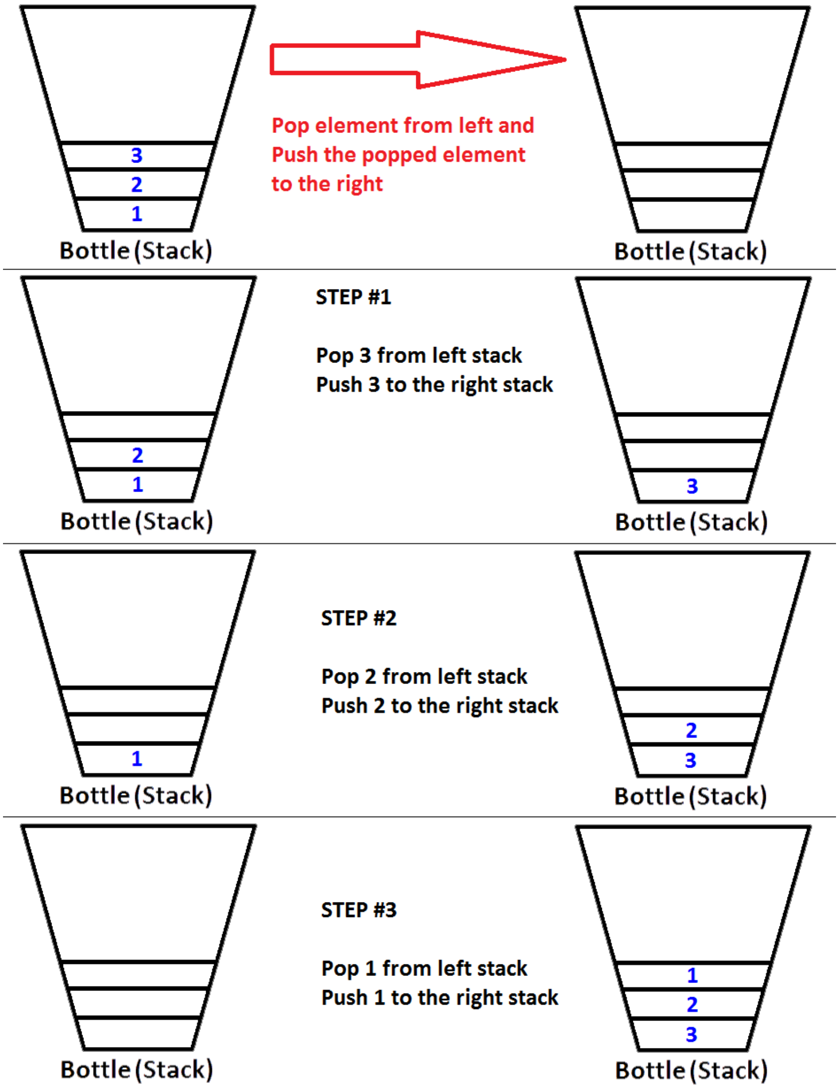
If we push integers 1,2,3 respectively, then 3 will be on the top of the stack. Because 1 will be pushed first, then 2 will be put on the top of 1. Lastly, 3 will be put on the top of the stack and latest state of our stack represented as a bottle will be as below;



Now we have our stack represented as a bottle is populated with values 3,2,1. And we want to reverse the stack so that the top element of the stack will be 1 and bottom element of the stack will be 3. What we can do ? We can take the bottle and hold it upside down so that all the values should reverse in order ?



Yes we can do that, but that's a bottle. To do the same process, we need to have a second stack that which is going to store the first stack elements in reverse order. Let's put our populated stack to the left and our new empty stack to the right. To reverse the order of the elements, we are going to pop each element from left stack, and push them to the right stack. You can see what happens as we do so on the image below;

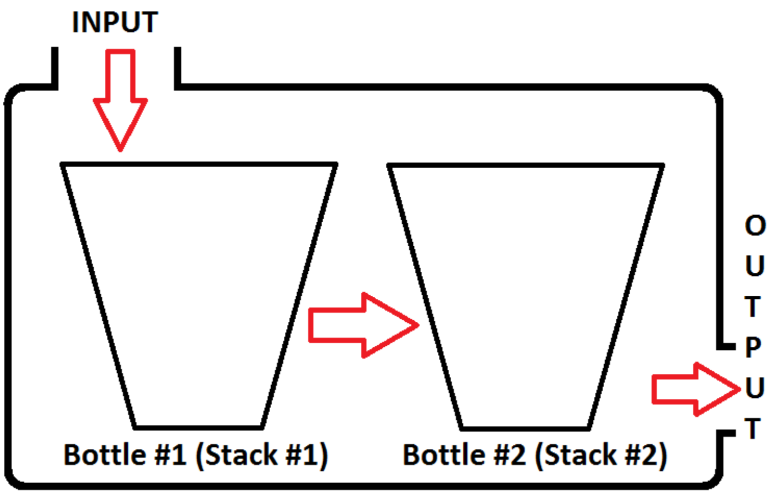


So we know how to reverse a stack.

B - Using Two Stacks As A Queue

On previous part, I've explained how can we reverse the order of stack elements. This was important, because if we push and pop elements to the stack, the output will be exactly in reverse order of a queue. Thinking on an example, let's push the array of integers `{1, 2, 3, 4, 5}` to a stack. If we pop the elements and print them until the stack is empty, we will get the array in the reverse order of pushing order, which will be `{5, 4, 3, 2, 1}` Remember that for the same input, if we dequeue the queue until the queue is empty, the output will be `{1, 2, 3, 4, 5}`. So it is obvious that for the same input order of elements, output of the queue is exactly reverse of the output of a stack. As we know how to reverse a stack using an extra stack, we can construct a queue using two stacks.

Our queue model will consist of two stacks. One stack will be used for `enqueue` operation (stack #1 on the left, will be called as Input Stack), another stack will be used for the `dequeue` operation (stack #2 on the right, will be called as Output Stack). Check out the image below;



Our pseudo-code is as below;

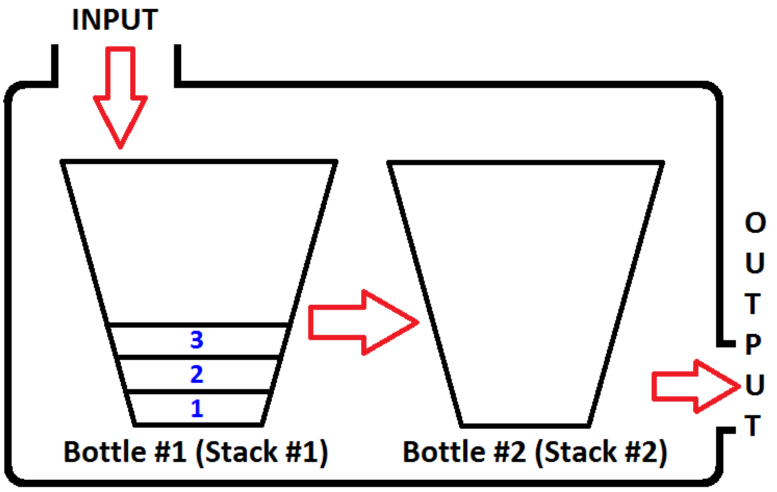
Enqueue Operation

```
Push every input element to the Input Stack
```

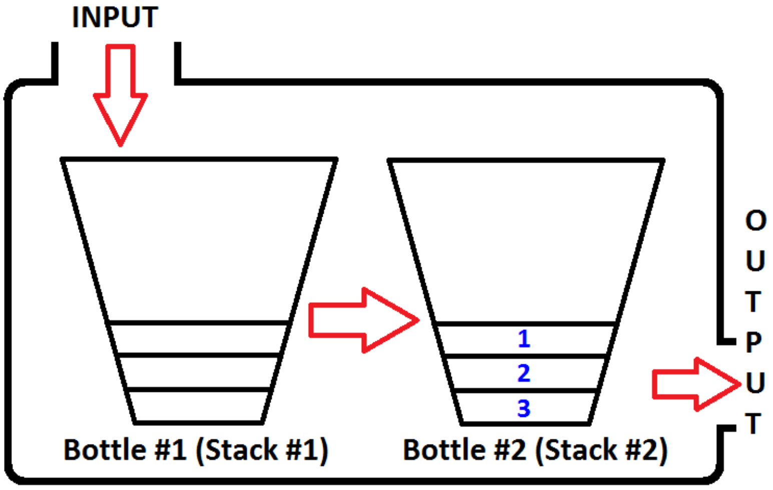
Dequeue Operation

```
If ( Output Stack is Empty)
    pop every element in the Input Stack
    and push them to the Output Stack until Input Stack is Empty

pop from Output Stack
```

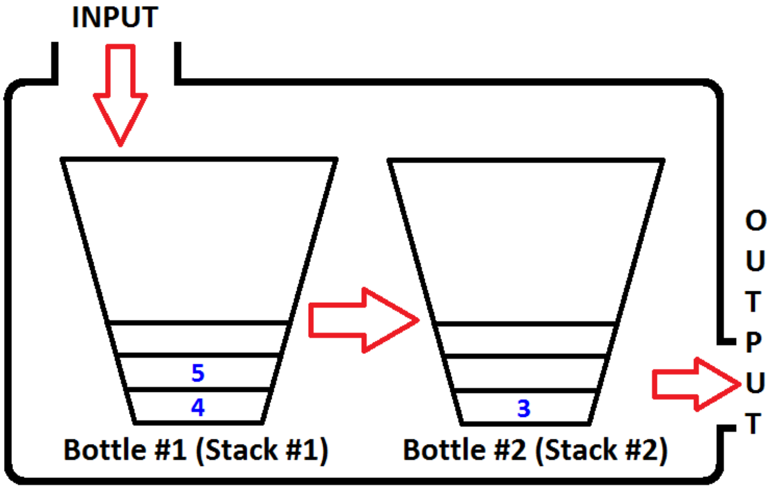
Let's enqueue the integers `{1, 2, 3}` respectively. Integers will be pushed on the **Input Stack** (**Stack #1**) which is located on the left;



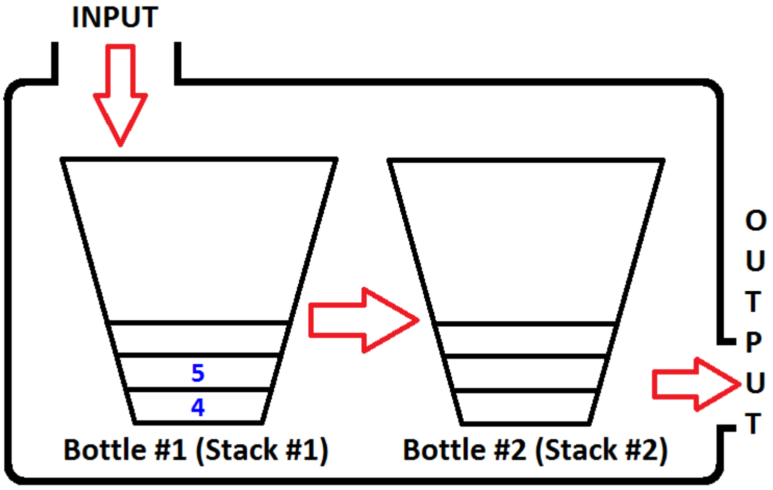
Then what will happen if we execute a dequeue operation? Whenever a dequeue operation is executed, queue is going to check if the Output Stack is empty or not(see the pseudo-code above) If the Output Stack is empty, then the Input Stack is going to be extracted on the output so the elements of Input Stack will be reversed. Before returning a value, the state of the queue will be as below;



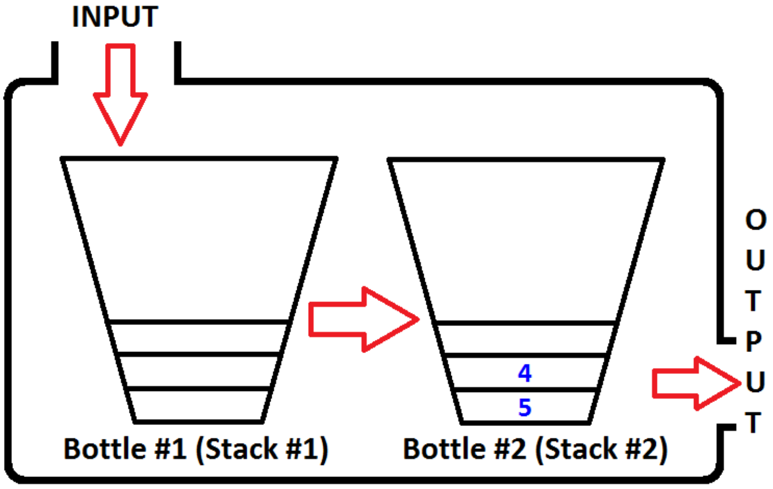
Check out the order of elements in the Output Stack (Stack #2). It's obvious that we can pop the elements from the Output Stack so that the output will be same as if we dequeued from a queue. Thus, if we execute two dequeue operations, first we will get `{1, 2}` respectively. Then element 3 will be the only element of the Output Stack, and the Input Stack will be empty. If we enqueue the elements 4 and 5, then the state of the queue will be as follows;



Now the Output Stack is not empty, and if we execute a dequeue operation, only 3 will be popped out from the Output Stack. Then the state will be seen as below;



Again, if we execute two more dequeue operations, on the first dequeue operation, queue will check if the Output Stack is empty, which is true. Then pop out the elements of the Input Stack and push them to the Output Stack unti the Input Stack is empty, then the state of the Queue will be as below;



Easy to see, the output of the two dequeue operations will be `{4, 5}`

C - Implementation Of Queue Constructed with Two Stacks

Here is an implementation in Java. I'm not going to use the existing implementation of Stack so the example here is going to reinvent the wheel;

C - 1) MyStack class : A Simple Stack Implementation

```
public class MyStack<T> {

    // inner generic Node class
    private class Node<T> {
        T data;
        Node<T> next;

        public Node(T data) {
            this.data = data;
        }
    }

    private Node<T> head;
    private int size;

    public void push(T e) {
        Node<T> newElem = new Node(e);

        if(head == null) {
            head = newElem;
        } else {
            newElem.next = head;
            head = newElem;     // new elem on the top of the stack
        }

        size++;
    }

    public T pop() {
        if(head == null)
            return null;

        T elem = head.data;
        head = head.next;   // top of the stack is head.next

        size--;

        return elem;
    }

    public int size() {
        return size;
    }

    public boolean isEmpty() {
        return size == 0;
    }

    public void printStack() {
        System.out.print("Stack: ");

        if(size == 0)
            System.out.print("Empty !");
        else
            for(Node<T> temp = head; temp != null; temp = temp.next)
                System.out.printf("%s ", temp.data);

        System.out.printf("\n");
    }
}
```

C - 2) MyQueue class : Queue Implementation Using Two Stacks

```
public class MyQueue<T> {

    private MyStack<T> inputStack;      // for enqueue
    private MyStack<T> outputStack;     // for dequeue
    private int size;

    public MyQueue() {
        inputStack = new MyStack<>();
        outputStack = new MyStack<>();
    }

    public void enqueue(T e) {
        inputStack.push(e);
        size++;
    }

    public T dequeue() {
        // fill out all the Input if output stack is empty
        if(outputStack.isEmpty())
            while(!inputStack.isEmpty())
                outputStack.push(inputStack.pop());

        T temp = null;
        if(!outputStack.isEmpty()) {
            temp = outputStack.pop();
            size--;
        }

        return temp;
    }

    public int size() {
        return size;
    }

    public boolean isEmpty() {
        return size == 0;
    }

}
```

C - 3) Demo Code

```
public class TestMyQueue {

    public static void main(String[] args) {
        MyQueue<Integer> queue = new MyQueue<>();

        // enqueue integers 1..3
        for(int i = 1; i <= 3; i++)
            queue.enqueue(i);

        // execute 2 dequeue operations
        for(int i = 0; i < 2; i++)
            System.out.println("Dequeued: " + queue.dequeue());

        // enqueue integers 4..5
        for(int i = 4; i <= 5; i++)
```

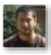
```
            queue.enqueue(i);

        // dequeue the rest
        while(!queue.isEmpty())
            System.out.println("Dequeued: " + queue.dequeue());
    }

}
```

C - 4) Sample Output

```
Dequeued: 1
Dequeued: 2
Dequeued: 3
Dequeued: 4
Dequeued: 5
```

edited Aug 23 '16 at 1:04

answered Aug 22 '16 at 23:01



Levent Divilioglu
3,896 ● 1 ● 23 ● 66

6 I would +1 this all day if I could. I couldn't make sense of how it was amortized constant time. Your illustrations really cleared things up, especially the part of leaving the remaining elements on the output stack, and only refilling when it empties. – Shane McQuillan Oct 10 '16 at 23:28

1 This really helped to prevent the timeout errors I was getting during pop. I was placing the elements back in the original stack but there was no need to do that. Kudos! – Pranit Bankar Oct 16 '16 at 21:29

1 All comments should be modeled after this one. – lolololol ol Dec 8 '16 at 17:36

2 I really didn't need a solution for this, just browsing... But when I see an answer like this I simply fall in love.. Great answer!!! – Maverick Feb 11 at 10:10

You can even simulate a queue using only one stack. The second (temporary) stack can be simulated by the call stack of recursive calls to the insert method.

The principle stays the same when inserting a new element into the queue:

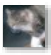
- You need to transfer elements from one stack to another temporary stack, to reverse their order.
- Then push the new element to be inserted, onto the temporary stack
- Then transfer the elements back to the original stack
- The new element will be on the bottom of the stack, and the oldest element is on top (first to be popped)

A Queue class using only one Stack, would be as follows:

```
public class SimulatedQueue<E> {
    private java.util.Stack<E> stack = new java.util.Stack<E>();

    public void insert(E elem) {
        if (!stack.empty()) {
            E topElem = stack.pop();
            insert(elem);
            stack.push(topElem);
        }
        else
            stack.push(elem);
    }

    public E remove() {
        return stack.pop();
    }
}
```

edited Sep 16 '08 at 21:10

answered Sep 16 '08 at 20:58



pythonquick
8,283 ● 6 ● 27 ● 27

6 Pretty elegant and nice use of recursion! – satyajit Jan 17 '11 at 3:43

45 Maybe the code looks elegant but it is very inefficient (O(n**2) insert) and it still has two stacks, one in the heap and one in the call stack, as @pythonquick points out. For a non-recursive algorithm, you can always grab one "extra" stack from the call stack in languages supporting recursion. – Antti Huima Apr 5 '11 at 17:59

3 @antti.huima is right! – Sobiaholic Oct 27 '12 at 13:08

1 @antti.huima And would you explain how this could be a quadratic insert?! From what I understand, insert does n pop and n push operations, so it's a perfectly linear O(n) algorithm. – LP_ Jan 17 '14 at 11:56

1 @LP_ it takes quadratic time O(n^2) to insert `n items` in the queue using the above data structure. the sum `(1 + 2 + 4 + 8 + .... + 2(n-1))` results in `~O(n^2)`. I hope you get the point. – Ankit Kumar Mar 15 '14 at 4:14

Brian's answer is the classically correct one. In fact, this is one of the best ways to implement persistent functional queues with amortized constant time. This is so because in functional programming we have a very nice persistent stack (linked list). By using two lists in the way

Brian describes, it is possible to implement a fast queue without requiring an obscene amount of copying.

As a minor aside, it is possible to prove that you can do *anything* with two stacks. This is because a two-stack operation completely fulfills the definition of a universal Turing machine. However, as Forth demonstrates, it isn't always easy. :-)

edited Sep 16 '08 at 4:01

answered Sep 16 '08 at 3:50
Daniel Spiewak
43k ● 10 ● 94 ● 116

3 en.wikipedia.org/wiki/… – user295190 Aug 23 '11 at 5:46

41 Where is Brian's answer ? – user1436489 Dec 19 '13 at 18:12

2 It was deleted because it wasn't very efficient. It copied elements between the two stacks – Dave L. Jun 26 '15 at 2:59

@DaveL.it was deleted, but I wish people would see it. If you want to understand more advanced solution, you first should learn inefficient but preferably easier to understand one. It will be abrupt someone to understand quick sort before consuming bubble sort, selection sort or insertion sort. – Teoman shipahi Jul 31 at 2:25

The time complexities would be worse, though. A good queue implementation does everything in constant time.

Edit

Not sure why my answer has been downvoted here. If we program, we care about time complexity, and using two standard stacks to make a queue is inefficient. It's a very valid and relevant point. If someone else feels the need to downvote this more, I would be interested to know why.

A little more detail: on why using two stacks is worse than just a queue: if you use two stacks, and someone calls dequeue while the outbox is empty, you need linear time to get to the bottom of the inbox (as you can see in Dave's code).

You can implement a queue as a singly-linked list (each element points to the next-inserted element), keeping an extra pointer to the last-inserted element for pushes (or making it a cyclic list). Implementing queue and dequeue on this data structure is very easy to do in constant time. That's worst-case constant time, not amortized. And, as the comments seem to ask for this clarification, worst-case constant time is strictly better than amortized constant time.

edited Apr 7 '11 at 6:56

answered Sep 16 '08 at 3:40
Tyler
23k ● 9 ● 72 ● 98

Not in the average case. Brian's answer describes a queue which would have amortized constant enqueue *and* dequeue operations. – Daniel Spiewak Sep 16 '08 at 3:53

That's true. You have average case & amortized time complexity the same. But the default is usually worst-case per-operation, and this is O(n) where n is the current size of the structure. – Tyler Sep 16 '08 at 4:51

1 Worst case can also be amortized. For example, mutable dynamic arrays (vectors) are usually considered to have constant insertion time, even though an expensive resize-and-copy operation is required every so often. – Daniel Spiewak Sep 16 '08 at 8:33

"Worst-case" and "amortized" are two different types of time complexity. It doesn't make sense to say that "worst-case can be amortized" -- if you could make the worst-case = the amortized, then this would be a significant improvement; you would just talk about worst-case, with no averaging. – Tyler Sep 16 '08 at 12:51

+1 for useful contribution to the subject. – thetoolman May 6 '12 at 20:16

Let queue to be implemented be q and stacks used to implement q be stack1 and stack2.

q can be implemented in **two** ways:

Method 1 (By making enQueue operation costly)

This method makes sure that newly entered element is always at the top of stack 1, so that deQueue operation just pops from stack1. To put the element at top of stack1, stack2 is used.

```
enQueue(q, x)
1) While stack1 is not empty, push everything from stack1 to stack2.
2) Push x to stack1 (assuming size of stacks is unlimited).
3) Push everything back to stack1.
deQueue(q)
1) If stack1 is empty then error
2) Pop an item from stack1 and return it.
```

Method 2 (By making deQueue operation costly)

In this method, in en-queue operation, the new element is entered at the top of stack1. In de-queue operation, if stack2 is empty then all the elements are moved to stack2 and finally top of stack2 is returned.

```
enQueue(q,  x)
 1) Push x to stack1 (assuming size of stacks is unlimited).

deQueue(q)
 1) If both stacks are empty then error.
 2) If stack2 is empty
   While stack1 is not empty, push everything from stack1 to stack2.
 3) Pop the element from stack2 and return it.
```

Method 2 is definitely better than method 1. Method 1 moves all the elements twice in enQueue operation, while method 2 (in deQueue operation) moves the elements once and moves elements only if stack2 empty.

edited Apr 13 '14 at 23:04
nevets1219
5,867 ● 3 ● 25 ● 44

answered Mar 30 '14 at 22:34
gandhi_rahul
110 ● 2 ● 8

None of the solutions I understood except for your method 2. I love the way you explain it with the enqueue and dequeue method with the steps. – theGreenCabbage Mar 15 '16 at 2:08

google.com/... – sat Mar 30 '16 at 16:47

Two stacks in the queue are defined as *stack1* and *stack2*.

Enqueue: The euqueued elements are always pushed into *stack1*

Dequeue: The top of *stack2* can be popped out since it is the first element inserted into queue when *stack2* is not empty. When *stack2* is empty, we pop all elements from *stack1* and push them into *stack2* one by one. The first element in a queue is pushed into the bottom of *stack1*. It can be popped out directly after popping and pushing operations since it is on the top of *stack2*.

The following is same C++ sample code:

```
template <typename T> class CQueue
{
public:
    CQueue(void);
    ~CQueue(void);

    void appendTail(const T& node);
    T deleteHead();

private:
    stack<T> stack1;
    stack<T> stack2;
};

template<typename T> void CQueue<T>::appendTail(const T& element) {
    stack1.push(element);
}

template<typename T> T CQueue<T>::deleteHead() {
    if(stack2.size()<= 0) {
        while(stack1.size()>0) {
            T& data = stack1.top();
            stack1.pop();
            stack2.push(data);
        }
    }


    if(stack2.size() == 0)
        throw new exception("queue is empty");


    T head = stack2.top();
    stack2.pop();


    return head;
}
```

This solution is borrowed from my blog. More detailed analysis with step-by-step operation simulations is available in my blog webpage.

edited Feb 27 '13 at 8:52
Andrew Barber
31.7k ● 13 ● 69 ● 104

answered Dec 29 '12 at 2:02
Harry He
712 ● 6 ● 10

You'll have to pop everything off the first stack to get the bottom element. Then put them all back onto the second stack for every "dequeue" operation.

edited Oct 23 '16 at 10:40
Witold Kaczurba
1,193 ● 8 ● 18

answered Sep 16 '08 at 4:26
user11055
198 ● 2 ● 3

2 Yes, you are right. I wonder, how you got so many down-votes. I have upvoted your answer – Binita Bharati Oct 21 '15 at 3:19 ✎

for c# developer here is the complete program :

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace QueueImplimentationUsingStack
{
    class Program
    {
        public class Stack<T>
        {
            public int size;
            public Node<T> head;
            public void Push(T data)
            {
                Node<T> node = new Node<T>();
                node.data = data;
                if (head == null)
                    head = node;
                else
                {
                    node.link = head;
                    head = node;
                }
                size++;
                Display();
            }
            public Node<T> Pop()
            {
                if (head == null)
                    return null;
                else
                {
                    Node<T> temp = head;
                    //temp.link = null;
                    head = head.link;
                    size--;
                    Display();
                    return temp;
                }
            }
            public void Display()
            {
                if (size == 0)
                    Console.WriteLine("Empty");
                else
                {
                    Console.Clear();
                    Node<T> temp = head;
                    while (temp!= null)
                    {
                        Console.WriteLine(temp.data);
                        temp = temp.link;
                    }
                }
            }
        }

        public class Queue<T>
        {
            public int size;
            public Stack<T> inbox;
            public Stack<T> outbox;
            public Queue()
            {
                inbox = new Stack<T>();
                outbox = new Stack<T>();
            }
            public void EnQueue(T data)
            {
                inbox.Push(data);
                size++;
            }
            public Node<T> DeQueue()
            {
                if (outbox.size == 0)
                {
                    while (inbox.size != 0)
                    {
                        outbox.Push(inbox.Pop().data);
                    }
                }
                Node<T> temp = new Node<T>();
                if (outbox.size != 0)
                {
                    temp = outbox.Pop();
                    size--;
                }
                return temp;
            }

        }
        public class Node<T>
        {
            public T data;
            public Node<T> link;
        }

        static void Main(string[] args)
        {
            Queue<int> q = new Queue<int>();
            for (int i = 1; i <= 3; i++)
                q.EnQueue(i);
           // q.Display();
            for (int i = 1; i < 3; i++)
                q.DeQueue();
            //q.Display();
            Console.ReadKey();
        }
    }
}
```

answered Nov 15 '16 at 5:55
Jaydeep Shil
565 ●4 ●13

```
public class QueueUsingStacks<T>
{
    private LinkedListStack<T> stack1;
    private LinkedListStack<T> stack2;

    public QueueUsingStacks()
    {
        stack1=new LinkedListStack<T>();
        stack2 = new LinkedListStack<T>();

    }
    public void Copy(LinkedListStack<T> source,LinkedListStack<T> dest )
    {
        while(source.Head!=null)
        {
            dest.Push(source.Head.Data);
            source.Head = source.Head.Next;
        }
    }
    public void Enqueue(T entry)
    {

       stack1.Push(entry);
    }
    public T Dequeue()
    {
        T obj;
        if (stack2 != null)
        {
            Copy(stack1, stack2);
             obj = stack2.Pop();
            Copy(stack2, stack1);
        }
        else
        {
            throw new Exception("Stack is empty");
        }
        return obj;
    }

    public void Display()
    {
        stack1.Display();
    }


}
```

For every enqueue operation, we add to the top of the stack1. For every dequeue, we empty the content's of stack1 into stack2, and remove the element at top of the stack.Time complexity is O(n) for dequeue, as we have to copy the stack1 to stack2. time complexity of enqueue is the same as a regular stack

edited Jun 19 '12 at 11:22
Matthias
3,410 ●4 ●34 ●71

answered Apr 11 '12 at 17:54
PradGar
7 ●2

This code is inefficient (unnecessary copying) and broken: `if (stack2 != null)` is always true because `stack2` is instantiated in the constructor. – melpomene Aug 6 '16 at 8:56

```
// Two stacks s1 Original and s2 as Temp one
    private Stack<Integer> s1 = new Stack<Integer>();
    private Stack<Integer> s2 = new Stack<Integer>();

    /*
     * Here we insert the data into the stack and if data all ready exist on
     * stack than we copy the entire stack s1 to s2 recursively and push the new
     * element data onto s1 and than again recursively call the s2 to pop on s1.
     *
     * Note here we can use either way ie We can keep pushing on s1 and than
     * while popping we can remove the first element from s2 by copying
     * recursively the data and removing the first index element.
     */
    public void insert( int data )
    {
        if( s1.size() == 0 )
        {
            s1.push( data );
        }
        else
        {
            while( !s1.isEmpty() )
            {
                s2.push( s1.pop() );
            }
            s1.push( data );
            while( !s2.isEmpty() )
            {
                s1.push( s2.pop() );
            }
        }
    }

    public void remove()
    {
        if( s1.isEmpty() )
        {
            System.out.println( "Empty" );
        }
        else
        {
            s1.pop();

        }
    }
```

answered Oct 13 '15 at 6:49

imvp
97 ● 1 ● 9

I'll answer this question in Go because Go does not have a rich a lot of collections in its standard library.

Since a stack is really easy to implement I thought I'd try and use two stacks to accomplish a double ended queue. To better understand how I arrived at my answer I've split the implementation in two parts, the first part is hopefully easier to understand but it's incomplete.

```
type IntQueue struct {
    front       []int
    back        []int
}

func (q *IntQueue) PushFront(v int) {
    q.front = append(q.front, v)
}

func (q *IntQueue) Front() int {
    if len(q.front) > 0 {
        return q.front[len(q.front)-1]
    } else {
        return q.back[0]
    }
}

func (q *IntQueue) PopFront() {
    if len(q.front) > 0 {
        q.front = q.front[:len(q.front)-1]
    } else {
        q.back = q.back[1:]
    }
}

func (q *IntQueue) PushBack(v int) {
    q.back = append(q.back, v)
}

func (q *IntQueue) Back() int {
    if len(q.back) > 0 {
        return q.back[len(q.back)-1]
    } else {
        return q.front[0]
    }
}

func (q *IntQueue) PopBack() {
    if len(q.back) > 0 {
        q.back = q.back[:len(q.back)-1]
    } else {
        q.front = q.front[1:]
    }
}
```

It's basically two stacks where we allow the bottom of the stacks to be manipulated by each other. I've also used the STL naming conventions, where the traditional push, pop, peek operations of a stack have a front/back prefix whether they refer to the front or back of the queue.

The issue with the above code is that it doesn't use memory very efficiently. Actually, it grows endlessly until you run out of space. That's really bad. The fix for this is to simply reuse the bottom of the stack space whenever possible. We have to introduce an offset to track this since a slice in Go cannot grow in the front once shrunk.

```
type IntQueue struct {
    front       []int
    frontOffset int
    back        []int
    backOffset  int
}

func (q *IntQueue) PushFront(v int) {
    if q.backOffset > 0 {
        i := q.backOffset - 1
        q.back[i] = v
        q.backOffset = i
    } else {
        q.front = append(q.front, v)
    }
}

func (q *IntQueue) Front() int {
    if len(q.front) > 0 {
        return q.front[len(q.front)-1]
    } else {
        return q.back[q.backOffset]
    }
}

func (q *IntQueue) PopFront() {
    if len(q.front) > 0 {
        q.front = q.front[:len(q.front)-1]
    } else {
        if len(q.back) > 0 {
            q.backOffset++
        } else {
            panic("Cannot pop front of empty queue.")
        }
    }
}

func (q *IntQueue) PushBack(v int) {
    if q.frontOffset > 0 {
        i := q.frontOffset - 1
        q.front[i] = v
        q.frontOffset = i
    } else {
        q.back = append(q.back, v)
    }
}

func (q *IntQueue) Back() int {
    if len(q.back) > 0 {
        return q.back[len(q.back)-1]
    } else {
        return q.front[q.frontOffset]
    }
}

func (q *IntQueue) PopBack() {
    if len(q.back) > 0 {
        q.back = q.back[:len(q.back)-1]
    } else {
        if len(q.front) > 0 {
            q.frontOffset++
        } else {
            panic("Cannot pop back of empty queue.")
        }
    }
}
```

It's a lot of small functions but of the 6 functions 3 of them are just mirrors of the other.

answered May 22 '16 at 16:08

John Leidegren
32.8k ● 13 ● 89 ● 123

You're using arrays here. I don't see where your stacks are. – melpomene Aug 6 '16 at 8:55

@melpomene OK, if you take a closer look you'll notice that the only operations that I'm performing is adding/removing of the last element in the array. In other words, pushing and popping. For all intent and purposes these are stacks but implemented using arrays. – John Leidegren Aug 6 '16 at 10:22

@melpomene Actually, that's only half right, I am assuming doubled ended stacks. I am allowing the stack to be modified in a non standard way from bottom up under certain conditions. – John Leidegren Aug 6 '16 at 10:24

A solution in c#

```
public class Queue<T> where T : class
    {
        private Stack<T> input = new Stack<T>();
        private Stack<T> output = new Stack<T>();
        public void Enqueue(T t)
        {
            input.Push(t);
        }

        public T Dequeue()
        {
            if (output.Count == 0)
            {
                while (input.Count != 0)
                {
                    output.Push(input.Pop());
                }
            }
            return output.Pop();
        }
}
```

answered May 31 at 0:08
Santhosh
469 ● 2 ● 10

here is my solution in java using linkedlist.

```
class queue<T>{
static class Node<T>{
    private T data;
    private Node<T> next;
    Node(T data){
        this.data = data;
        next = null;
    }
}
Node firstTop;
Node secondTop;

void push(T data){
    Node temp = new Node(data);
    temp.next = firstTop;
    firstTop = temp;
}

void pop(){
    if(firstTop == null){
        return;
    }
    Node temp = firstTop;
    while(temp != null){
        Node temp1 = new Node(temp.data);
        temp1.next = secondTop;
        secondTop = temp1;
        temp = temp.next;
    }
    secondTop = secondTop.next;
    firstTop = null;
    while(secondTop != null){
        Node temp3 = new Node(secondTop.data);
        temp3.next = firstTop;
        firstTop = temp3;
        secondTop = secondTop.next;
    }
}
```

}

Note : In this case, pop operation is very time consuming. So i won't suggest to create a queue using two stack.

answered Sep 24 at 13:32
Irshad ck
34 ● 2

Queue implementation using two java.util.Stack objects:

```
public final class QueueUsingStacks<E> {

        private final Stack<E> iStack = new Stack<>();
        private final Stack<E> oStack = new Stack<>();

        public void enqueue(E e) {
            iStack.push(e);
        }

        public E dequeue() {
            if (oStack.isEmpty()) {
                if (iStack.isEmpty()) {
                    throw new NoSuchElementException("No elements present in
Queue");
                }
                while (!iStack.isEmpty()) {
                    oStack.push(iStack.pop());
                }
            }
            return oStack.pop();
        }

        public boolean isEmpty() {
            if (oStack.isEmpty() && iStack.isEmpty()) {
                return true;
            }
            return false;
        }

        public int size() {
            return iStack.size() + oStack.size();
        }

}
```

answered Aug 6 '16 at 8:04

realPK
410 ● 6 ● 16

2 This code is functionally identical to the answer by Dave L. It adds nothing new, not even an explanation. – melpomene Aug 6 '16 at 8:59

It adds isEmpty() and size() methods along with basic exception handling. I will edit to add explanation. – realPK Aug 6 '16 at 15:23

1 No one asked for those extra methods, and they're trivial (one line each): `return inbox.isEmpty() && outbox.isEmpty()` and `return inbox.size() + outbox.size()`, respectively. Dave L.'s code already throws an exception when you dequeue from an empty queue. The original question wasn't even about Java; it was about data structures / algorithms in general. The Java implementation was just an additional illustration. – melpomene Aug 6 '16 at 17:07

1 This is a great source for people looking to understand how to build queue from two stacks, the diagrams most definitely helped me more than reading Dave's answer. – Kemal Tezer Dilsiz Oct 3 '16 at 5:34

@melpomene: Its not about methods being trivial but of need. Queue interface in Java extends those methods from Collection interface because they are needed. – realPK Mar 4 at 23:37

protected by Community ♦ May 18 '13 at 4:09

Thank you for your interest in this question. Because it has attracted low-quality or spam answers that had to be removed, posting an answer now requires 10 reputation on this site (the association bonus does not count).

Would you like to answer one of these unanswered questions instead?