March 17, 2022

IMPLEMENTING QUEUE USING STACKS

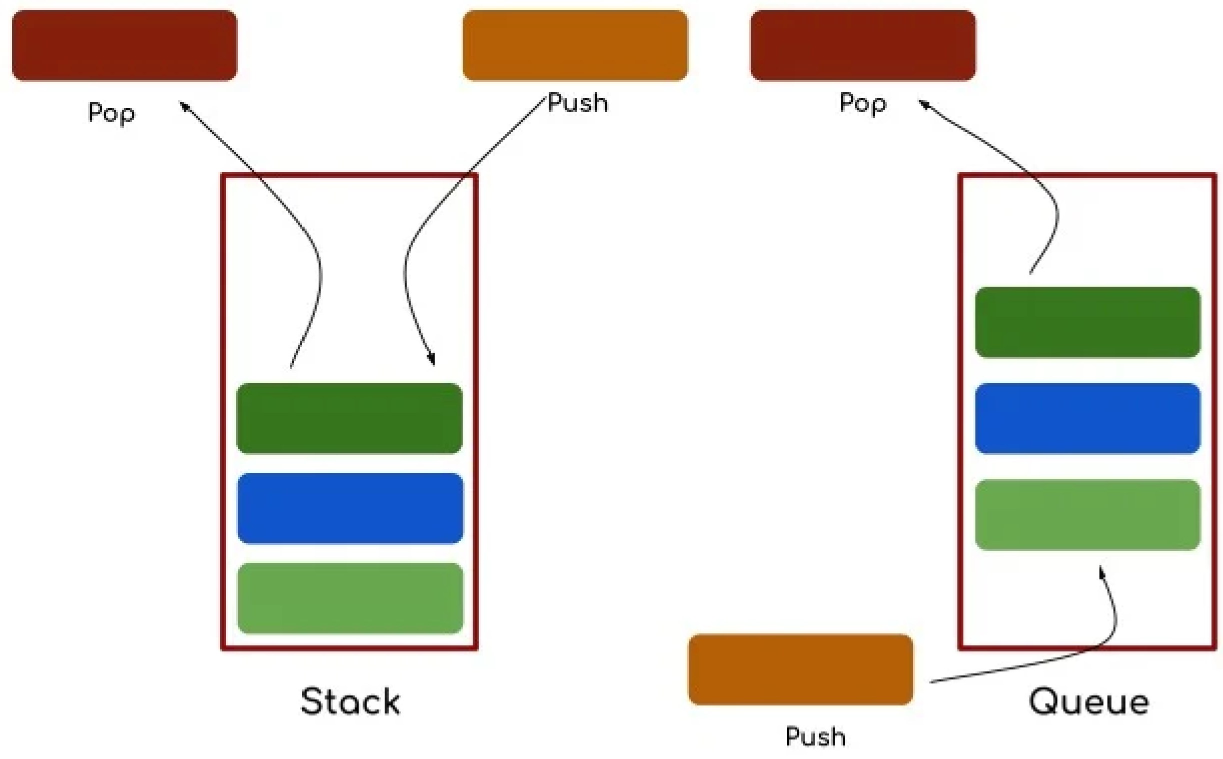
Nikhil Kumar Singh
Vrishchik

DURATION
9min

CATEGORIES
Competitive Programming
How-To
Tutorials

TAGS
C++

A stack can be a linear data structure that follows Last In First Out (LIFO), in which iteration is performed at the forepart and removal is finished from the backside. Queue can be a linear data structure that follows First In First Out (FIFO), where iteration is performed at the side and removal is done from the face.



For our problem of implementing a queue using stack, we require two stacks. There are two variations

- Making the pushing operation costlier
- Making the popping operation costlier

APPROACH #1: MAKING THE PUSHING OPERATION COSTLIER

In this approach, the enqueue(push) operation has more complexity than the dequeue(pop) operation, O(n) and O(1) respectively. As stated, we need two stacks for this, let them be S1 and S2.

In the case of enqueue, we pop all the values from S1 and push them into S2. Then, we insert the new value into S1 and again pop all from S2 and push them into S2, resulting in a linear complexity O(n). For the dequeue operation, we just pop the top element from S which results in a constant complexity, O(1).

Code:

```
#include<iostream>

#include<stack>

using namespace std;

class Queue {
 private:
  stack < int > s1, s2;
 public:
  // Enqueue function of Queue
  void enqueue(int val) {

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

   s1.push(val);

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

   cout << "enqueued :" << val << endl;
  }

 // Dequeue function of Queue
 void dequeue() {

  if (!s1.empty()) {
```

```cpp
      cout << "dequeued :" << s1.top() << endl;
      s1.pop();
    } else {
      cout << "Underflow " << endl;
    }
  }
  // Front
  void front() {
    if (!s1.empty()) {
      cout << "top :" << s1.top() << endl;
    } else {
      cout << "Underflow " << endl;
    }
  }

};

int main() {
  Queue q;
  q.enqueue(6);
  q.enqueue(3);
  q.enqueue(9);
  q.dequeue();
  q.front();
  q.enqueue(12);
  q.front();
  q.dequeue();
  q.dequeue();
  q.dequeue();

}
```

Output:

```
enqueued :6
enqueued :3
enqueued :9
dequeued :6
top :3
enqueued :12
top :3
dequeued :3
```

dequeued :9

dequeued :12

APPROACH #2: MAKING THE POPPING OPERATION COSTLIER

In this approach, the enqueue(push) operation has less complexity than the dequeue(pop) operation, O(1) and O(n) respectively. As stated we need two stacks for this, let them be S1 and S2.

For the enqueue operation, we just push to the top of S1, which results in a constant complexity, O(1).
In the case of dequeue we insert the new element and then pop all the values from S1 and push them into S2. Then, pop the topmost element from S2 then pop all from S2 and push them into S1, resulting in a linear complexity O(n).

Code:

```
 #include<iostream>

#include<stack>

using namespace std;

class Queue {
  private:
    stack < int > s1, s2;
  public:
    // Enqueue function of Queue
    void enqueue(int val) {
      s1.push(val);
      cout << "enqueued :" << val << endl;
    }

  // Dequeue function of Queue
  void dequeue() {
    if (s2.empty()) {
      while (!s1.empty()) {
        s2.push(s1.top());
        s1.pop();
      }
    }
    if (!s2.empty()) {
      cout << "dequeued :" << s2.top() << endl;
      s2.pop();
```

? 支持

```
    } else {
      cout << "Underflow " << endl;
    }
  }
  // Front
  void front() {
    if (s2.empty()) {
      while (!s1.empty()) {
        s2.push(s1.top());
        s1.pop();
      }
    }
    if (!s2.empty()) {
      cout << "top :" << s2.top() << endl;
    } else {
      cout << "Underflow " << endl;
    }
  }

};

int main() {
  Queue q;
  q.enqueue(6);
  q.enqueue(3);
  q.enqueue(9);
  q.dequeue();
  q.front();
  q.enqueue(12);
  q.front();
  q.dequeue();
  q.dequeue();
  q.dequeue();

}
```

Output:

```
enqueued :6
enqueued :3
enqueued :9
dequeued :6
top :3
```

支持

```
enqueued :12
top :3
dequeued :3
dequeued :9
dequeued :12
```

0 **0**