Stacks

Given an input sequence of integers 1, 2, 3, 4, 5, 6 and only three operations on a stack:
1. C: Copy next input directly to output list.
3. P: Pop stack and send popped integer to output.

For example, the sequence of operations: ‘SSCCPSPPP’ produces the output ‘453621’.
Notice that the no. of pops (P) must be equal to the no. of pushes (S) and that the sum of 
pops (P) and copies (C) must be equal to 6.
Write the sequences of operations needed to produce the following outputs:
(a) 123456 (b) 654132 (c) 143526 (d) 123654 (e) 243561 (f) 345621

Answer:

The input sequence is like a wagon train. At the input, the wagons are numbered 1 2 3...
The stack is like a garage; we use it to obtain a certain “permutation” of the wagons on 
the output. For example for (f):
‘SSCCCP’ produces 345621

The exercise shows that a stack can be used to produce permutations of an input 
sequence. Can we use a queue + stack to permute an input sequence? If yes, work out your own example.

Convert the following infix expressions to postfix notation:
A + B * C  A + B – C  (A + B) * C  (A * B / C) / (A – D)

Answer:
The first will be ABC*+. Work out the others.

Evaluate the postfix expression  A B C D - * +, where A = 25, B = 2, C = 18 and D = 13.

Answer:
The stack will produce A+B*(C-D) = 35
In the following exercises, assume that you are developing an “application” using the “Stackt” class. You have no access to the implementation of the class.

Write a function `GetBottom` to retrieve the bottom element from a non-empty stack of characters. The procedure should leave the stack unchanged.

Write a recursive function `Append(S,U)` to append a stack (U) on top of a stack (S) so that U's top is on the top, leaving U empty.

**Answer:**
```c++
void Append(Stackt<Type> & s, Stackt<Type> & u)
{
    Type el;
    if(!u.stackIsEmpty())
    {
        u.pop(el);
        Append(s, u);
        s.push(el);
    }
}
```

Write a function `RemoveBlanks` to remove all blanks from a stack of characters, leaving the stack otherwise unchanged. What is the complexity of this function?

Write a function `SwapStack` to exchange the top two elements of the stack, leaving the stack otherwise unchanged.

**Answer:**
```c++
void SwapStack(Stackt<char> & s)
{
    char el, temp;
    if(!s.stackIsEmpty())
    {
        s.pop(el);
        if(!s.stackIsEmpty())
        {
            s.pop(temp);
            s.push(el);
            s.push(temp);
        }
        else s.push(el);
    }
}
```

Write a function `AddTop` to replace the top two elements of a nonempty stack of numbers with their sum. If the stack contains one element, we leave the stack unchanged.
Write a recursive function \texttt{AddStack} to replace the elements of a nonempty stack of numbers with their sum. If the stack contains one element, we leave the stack unchanged. \textbf{Answer:}

```c
void AddStack(Stackt <int> & s)
{
    int el1, el2;

    if(!s.stackIsEmpty())
    {
        s.pop(el1);
        if(!s.stackIsEmpty())
        {
            s.pop(el2);
            s.push(el1 + el2);
            AddStack(s);
        }
        else s.push(el1);
    }
}
```

Write a function \texttt{EqualStacks} to return True if two stacks are identical. The function should leave the stacks unchanged.

\textbf{In the following, assume that you have access to the implementation of the “Stackt” class:}
- Overload the “==” operator to test if two stacks are identical. The operation should leave the stacks unchanged.
- Overload the “+” operator to append a stack on top of another stack.

\textbf{Answer for the == operator:}

```c
template <class Type>
bool Stackt<Type>::operator == (const Stackt<Type> & s)
{
    if (s.top != top) return false;

    for ( int i = 0; i <= top; i++)
        if (stack[i] != s.stack[i])
            return false;
    return true;
}
```
In the following exercises, assume that you are developing an “application” using the “Queue” class. You have no access to the implementation of the class.

Write a function **QueueRear** (**Q**) to retrieve the rear of a queue (**Q**), leaving the queue unchanged.

Write a function **NthElement** (**Q**) to return the \( n \)th element in a queue (**Q**), leaving the queue without that element.

**Answer:**

```cpp
Type NthElement(Queuet <Type> & Q, int n)
{
    Type el, nth;
    int len = Q.queueLength();
    for(int i=1; i<=len; i++)
    {
        Q.dequeue(el);
        if(i != n)
            Q.enqueue(el);
        else
            nth = el;
    }
    return nth;
}
```

Write a boolean function **EqualQueues** (**Q1**, **Q2**) which receives two queues **Q1** and **Q2** and returns True if they are identical and False otherwise. The function should leave the queues unchanged.

A queue (**Q**) contains an even number of elements. Write a function **Split** (**Q**, **Q1**, **Q2**) to copy the 1st half of (**Q**) into (**Q1**) and the 2nd half into (**Q2**), leaving the original queue unchanged.
void Split(Queuet<int> & Q, Queuet<int> & Q1, Queuet<int> & Q2)
{
    int len = Q.queueLength();
    int el;
    for(int i=1; i <= len/2; i++) {
        Q.dequeue(el);
        Q1.enqueue(el);
        Q.enqueue(el);
    }
    for(i=1; i <= len/2; i++) {
        Q.dequeue(el);
        Q2.enqueue(el);
        Q.enqueue(el);
    }
}

Write a function SwapQueue (Q) to exchange the front two elements of the queue (Q),
leaving the queue otherwise unchanged.

Write a recursive function Append (Q1,Q2) to append a queue (Q2) to the end of a
queue (Q1), leaving Q2 empty.
Answer:
void Append(Queuet<int> & Q1, Queuet<int> & Q2)
{
    int el;
    if(!Q2.queueIsEmpty())
    {
        Q2.dequeue(el);
        Q1.enqueue(el);
        Append(Q1, Q2);
    }
}

Write a recursive function Reverse (Q) to reverse the order of the elements in a queue
(Q).

Write a function PositionToQueue (Q, Sub, S) to place into queue (Q) the starting
positions, in order, of every occurrence of substring (sub) in the string (S).
Answer:
void PositionToQueue(Queuet<int> & Q, string s, string sub)
{
    int pos = s.find(sub);
    int i = 0;
while(pos>=0 && pos < s.length())
{
    Q.enqueue(pos + i * sub.length());
    s.erase(pos, sub.length());
    pos = s.find(sub);
    i++;
}

In the following, assume that you have access to the implementation of the “Queuet” class:

- Overload the “==” operator to test if two queues are identical. The operation should leave the queues unchanged.
- Overload the “+” operator to append a queue to the rear of another queue.