ECE 250 *Algorithms and Data Structure*
Department of Electrical and Computer Engineering
University of Waterloo


**3.2*a*** What is the state of the stack after the following sequence of pushes and pops:

```
Stack<int> s;
s.push( 3 );
s.push( 5 );
s.push( 2 );
s.push( 15 );
s.push( 42 );
s.pop();
s.pop();
s.push( 14 );
s.push( 7 );
s.pop();
s.push( 9 );
s.pop();
s.pop();
s.push( 51 );
s.pop();
s.pop();
```

**3.2*b*** Suppose the numbers 0, 1, 2, …, 9 were pushed onto a stack in that order, but that pops occurred at random points between the various pushes. The following is a valid sequence in which the values in the stack could have been popped:

3, 2, 6, 5, 7, 4, 1, 0, 9, 8

Explain why it is not possible that

3, 2, 6, 4, 7, 5, 1, 0, 9, 8

is a valid sequence in which the values could have been popped off the stack.

**3.2*c*** Suppose that the `Stack` class uses the class `Single_list` and we want to move the contents of one stack onto another stack. Because the `Stack` class is not a friend of the `Single_list` class (and it would be foolish to allow this), we need a new `push_front( Single_list & )` function that moves the contents of the argument onto the front of the current linked list in $\Theta(1)$ time while emptying the argument. In such a way, we could do the following:

```
template <typename Type>
void Stack<Type>::push_front( Stack<Type> const &s ) {
    list.push_front( s.list );
}
```

Implement the function `push_front`:

```
template <typename Type>
void Single_list<Type>::push_front( Single_list<Type> &list ) {




}
```

**3.2*d*** Why must the argument in 3.2*c* be passed by reference, but not passed by constant reference?

**3.2*e*** Suppose we wanted to implement a member function `clear()` which removed all the current entries from a stack class that uses a `Single_list` internally. How would you implement such a function?

```
template <typename Type>
void Stack<Type>::clear() {




}
```

**3.2*f*** Consider the undo and redo operations or forward and back operations on a browser. While it is likely more obvious that operations to undo or pages to go back to may be stored using a stack, what is the behaviour of the redo or page forward operations? How is it related to being a stack? Are there times at which the redo or forward operations stored in the stack are cleared (see 3.2.3.1*c*).

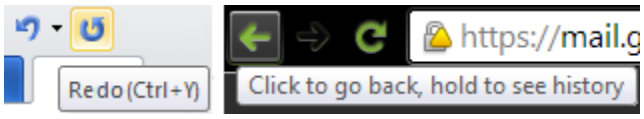


Figure 1. The undo/redo buttons of an editing application
and the back/forward buttons of a web browser.

**3.2*g*** Suppose we wanted to implement a member function `clear()` which removed all the current entries from a stack class that uses an array internally. How would you implement such a function?

```
template <typename Type>
void Stack<Type>::empty() {




}
```

**3.2*h*** You have written a `void double_capacity()` function for your Stack class using an array. This function doubles the size of the array, copies the old values over, and frees the memory for the old array.

Update the push member function to call this function if the stack is full given the member variables `array_size` and `array_capacity` as described in class.

```
template <typename Type>
void Stack<Type>::push( Type const &obj ) {


    array[stack_size] = obj;


    ++stack_size;


}
```

**3.2*i*** When we use an internal array to store the values of a stack, one strategy for when the array is full is to double the size of the array. This may lead to a very large array which may not always be necessary. One strategy would be to halve the size of the array if the capacity of the array drops below a certain value. Explain why it would be a bad idea to halve the size of the array if the array is half empty. Justify your answer using algorithm analysis with the appropriate Landau symbols.

**3.2*j*** Given the characteristics of integer and floating-point operations, which of the following conditions would you use to test whether or not *a* is less than or equal to a third of *b*, and explain the issues that may arise if you use one of the other two.

```
if ( a >= b / 3 ) { // ...

if ( a >= b / 3.0 ) { // ...

if ( 3*a >= b ) { // ...
```

**3.2*k*** Suppose that we increase the size of an array by the scalar multiple of 1.3 each time the array becomes full. What is an estimate of the number of copies made per insertion? What is the worst case for number of unused memory locations?

**3.2*l*** Suppose that we increase the size of an array by the fixed amount of 13 entries each time the array becomes full. What is an estimate of the number of copies made per insertion? What is the worst case for number of unused memory locations?

**3.2*m*** Do the tags in the following code fragment correctly match as required byXML (assuming all the tags are valid)?

```
<a><b><c>d</c><e>f<g><g>h</g>i<j>k</j>l</g></e></b><m>n</m></a>
```

**3.2*n*** Given the following XTML fragment, what is the state of the stack (recording the opening tags that have not yet been matched) at the end of this fragment?

```
<xhtml>
<body>
<p>1b. If lg(<i>n</i>) = log<sub>2</sub>(<i>n</i>) = <i>x</i> then what
is log<sub>16</sub>(<i>n</i>) in terms of <i>x</i>?</p>
<p>1c. Show that lg(2<i>n</i>) = 1 + lg(<i>n</i>) and that
ln(2<i>n</i>) = ln(2) + ln(<i>n
```

**3.2*o*** At which character would the parser indicate an error in the following code fragments?

```
n[(m + 3)*a - (b - 7)] = c[(f[g(3)+4*h(5)*(4 + d)] – 7*e(5 + 4*i) + 14];
n[(m - 3)*a - (4 - b)] = c[(f[g(3)+4*(5 - h)*(4 - d)] – 7*e(5 + i)) + 14;
n[(1 + m)*a - b)] = c[(f[g(3)+4*(5 + h)*(4 + d)] – 7*e(5 + 4*i)) + 14];
```

**3.2*p*** Given the following C++ fragment, what is the state of the stack (recording the opening delimiters—`(`, `[`, `{`, `<`—that have not yet been matched) at the end of this fragment?

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5 const int M = 30;
6 int N = 1024;
7 int parent[N], height[N], depth[N];
8 int maxheights[100];
9
10 for ( int i = 0; i < 100; ++i ) {
11 maxheights[i] = 0;
12 }
13
14 double minmeandepth = 1e300;
15 double meandepths = 0;
16 double maxmeandepth = 0;
17
18 for ( int j = 0; j < M; ++j ) {
19 for ( int i = 0; i < N; ++i ) {
20 parent[i] = -1;
21 height[i] = 0;
22 }
23
24 for ( int i = 0; i < N - 1; ++i ) {
25 while ( true ) {
26 int p1 = rand() & (N - 1);
27 int p2 = rand() & (N - 1);
28
28 int s1 = p1;
30 int s2 = p2;
31
32 while ( parent[s1
```

**3.2*q*** Suppose that each call of the function factorial occupies 100 bytes on the call stack. What is the maximum size of the stack when this function is called with the argument 7:

```
int factorial( int n ) {
    return (n <= 1) ? 1 : factorial( n - 1 );
}
```

What is the total number of function calls made, including the initial call with the argument 7?

**3.2*r*** The $n^{th}$ Fibonacci number is defined as the sum of the two previous Fibonacci numbers where the $0^{th}$ and $1^{st}$ Fibonacci numbers are defined as 1. What is the maximum size of the stack when this function is called with the argument 4:

```
int Fibonacci( int n ) {
    return (n <= 1) ? 1 : Fibonacci( n - 1 ) + Fibonacci( n - 2 );
}
```

What is the total number of function calls made, including the initial call with the argument 4?

**3.2*s*** Which of the following are valid and complete reverse Polish expressions? The operators + and * represent the standard binary addition and multiplication operations, respectively.

```
1 2 + * 4 5 * 6 + +
1 2 3 + * 4 5 * + +
1 2 + 3 * 4 5 * 6 +
```

**3.2*t*** Evaluate the following expressions that are written using reverse Polish notation:

```
1 2 3 + * 4 5 * 6 + +
1 2 3 + * 4 5 * + 6 +
1 2 + 3 * 4 5 * 6 + +
```

**3.2*u*** Suppose someone created a C parser that used reverse Polish notation and you were presented with the line of code:

```
a value . 3 4 s sin * + =
```

Here `sin` is a function of one argument and `.` is the member access operator, while `a` and `s` are variables. Return this program to sanity: how would this statement appear in the normal C++ programming language?