## Labs 39 - 49

*Lab 39 - Palindrome Test with Stacks*

Lambda expression for `transform`:

```
 auto myUpper = [](char c) -> char{ return toupper(c); };
 transform(str.begin(), str.end(), str.begin(), myUpper);
```

Stack *a* held *backward* order, stack *b* was sacrificial lamb, stack *c* held *forward* order. Test top elements of *a* and *c*, return false for any discrepancies.

*Lab 40 - Nested Delimiters Test with Stack*

Loop through string, push left delimiters onto stack. If right delimiter is found and matches top element of stack, pop stack. If stack is empty within loop, return false. If stack isn't empty by the end of the loop, return false.

*Lab 41 - Infix to Postfix*

*Lab 42 - Palindrome Test with Stack and Queue*

Stack holds backward string, queue holds forward string. If discrepancy is found between top elements, return false.

*Lab 43 - Queue Bucket Sort*

Make vector of queues and resize it to have 10 rows. Sort by ones, tens, etc. and push into proper queue. While the queue isn't empty, put elements back into vector and pop.

*Lab 44 - Test Palindrome with Deque*

Push string into deque. Test front and back elements and return false if discrepancy. If size is 1, return true.

*Lab 45 - BigInt Constructors, Equality, and Less Than*

Default constructor sets sign to ZERO and places one zero in the deque. Int constructor just puts int to_string in string constructor. String constructor puts BigInt in reverse. Sign starts as POSITIVE and changes if '-' is encountered. If digits is encountered, push back into deque. Checks back element to remove leading 0s, check if empty to avoid core dump. If deque is empty, set sign to ZERO and put '0' in deque. Copy constructor sets sign and digits of *this equal to other. To test for equality, test sign, then digits. To test less than, test size, then contents, then contents digit-by-digit.

*Lab 46 - BigInt Addition and Subtraction*

*Lab 47 - BigInt Multiplication*

*Lab 48 - Distribution with Priority Queue*

If copy is not empty, set current to top element. While copy is not empty, check if top element is equal to current and increment if so, else output current, change current to top, and reset count to 1. If copy size equals one, put top element to current and output.

*Lab 49 - Priority Queue Sort*

Put elements of vector into priority queue. While priority queue is not empty, put top element into vector and pop in reverse order.

### Chapter 19: Stacks

LIFO/FIFO adaptor. Can access and manipulate only the most recent element added to the stack. `stack` is a back-insertion sequence. It can be implemented as a `vector`, `deque` (default), or `list` since it needs the functions `back()`, `push_back()`, and `pop_back()`. It is defined in `<stack>`. A `stack` is designed for efficiency on the back end, and it possesses no indexing, iterators, or pointers.

Defining a `stack`:

```
 stack <char, vector<char>> cstack;
 stack <char, list<char>> cstack;
 stack <char> cstack;
```

`stack` Functions:

Constant: `top(), empty(), size(), push(), emplace(), pop(), swap()`

Linear: `(constructor), (destructor), operator=,` relations

### Chapter 19: Queues

FILO/LILO adaptor. Features efficiency on both ends. `queue` is a front-insertion sequence. It can be implemented as a `deque` (default) or `list` since it needs the functions `back()`, `front()`, `push_back()`, and `push_front()`. It is defined in `<queue>`. A `queue` is designed for efficiency on both ends. We can't use a `vector` because it has front insertion in O(N) time. It offers no indexing, iterators, or pointers.

Defining a `queue`:

```
 queue<char> symbols;
 queue<char, list<char>> delimiters;
```

`queue` Functions:

Constant: `front(), back(), empty(), size(), push(), emplace(), pop(), swap()`

Linear: `(constructor), (destructor), operator=,` relations

### Chapter 19: Deques

Indexed sequence container built for fast insertion and deletions at the start and end. A `deque` never has to move more than half of its elements for an insertion or deletion. The elements of a `deque` are not stored contiguously and instead as a sequence of individually allocated, fixed-size arrays; can be seen as a 2D array. Storage is added and removed as needed; expansion is cheaper than expansion of a `vector`. Allows for random access, iterators, pointers, and indexing.

Defining a `deque`:

```
 deque<char> cDeque;
```

`deque` Functions:

Constant: `get_allocator(), at(), operator[], front(), back(), begin + cbegin, end + cend, rbegin + crbegin, rend + crend, empty(), size(), max_size(), push_back(), emplace_back(), pop_back(), push_front(), emplace_front(), pop_front(), swap`

Linear: `(constructor), (destructor), operator=, assign(), shrink_to_fit(), clear(), insert(), emplace(), erase(), resize(), erase(), erase_if(),` relations

### Priority Queues:

A container adaptor that provides constant time lookup of the largest (by default) element at the expense of logarithmic insertion and extraction. It can be implemented as a `vector` (default) or a `deque` since it requires the functions `front(), push_back(), and pop_back()`. It is defined in `<queue>`. It offers no indexing, iterators, or pointers. There is no real agreement on what counts as high priority, so programmers can make their own *weak comparisons* to use.

Defining a `priority_queue`:

```
 priority_queue<int> a; // vector, less
 priority_queue<int>, deque<int> b; // deque, less
 priority_queue<int, vector<int>, greater<int>>; // greater
```

`priority_queue` Functions:

Constant: `top(), empty(), size(), swap()`

Logarithmic: `push(), emplace(), pop()`

Linear: `(constructor), (destructor), operator=`

**`make_heap`, `push_heap`, and `pop_heap`**

`make_heap` constructs a heap with the given comparison function. It runs on a complexity of O(N), or at most 3*distance(first, last) comparison. `push_heap` inserts a new element at the position *last-1*. It runs with a complexity of O(logN). `pop_heap` swaps the position in *first* and *last-1* and makes the subrange *[first, last-1]*. It runs in O(logN) time with at most 2*log(N) comparisons.