Array List Implementation:

Array Resize Strategy #1:

...total copies across all resizes: _________
...total number of insert operations: _________
...average (amortized) cost of copies per insert: _________

Array Resize Strategy #2:

...total copies across all resizes: _________
...total number of insert operations: _________
...average (amortized) cost of copies per insert: _________

Running Time:

<table>
<thead>
<tr>
<th></th>
<th>Singly Linked List</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert/Remove at <strong>front</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Insert after a <strong>given</strong> element</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remove after a <strong>given</strong> element</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Insert at <strong>arbitrary</strong> location</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remove at <strong>arbitrary</strong> location</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Stack ADT

Running Time:

Stack and Queue Implementations

**Stack.h**

```c++
#pragma once
#include <vector>
template <typename T>
class Stack {
public:
    void push(const T & d);
    T pop();
    bool isEmpty();
private:
    std::vector<T> list_;
};

#include "Stack.hpp"
```

**Stack.hpp**

```c++
template <typename T>
void Stack<T>::push(const T & d) {
    list_.push_back(d);
}

template <typename T>
T Stack<T>::pop() {
    T data = list_.back();
    list_.pop_back();
    return data;
}
```
Circler Queue

Example 1

```cpp
Queue<int> q;
q.enqueue(3);
q.enqueue(8);
q.enqueue(4);
q.dequeue();
q.enqueue(7);
q.dequeue();
q.dequeue();
q.enqueue(2);
q.enqueue(1);
q.enqueue(3);
q.enqueue(5);
q.enqueue(4);
q.enqueue(9);
```

Example 2

```cpp
Queue<char> q;
q.enqueue('m');
q.enqueue('o');
q.enqueue('n');
... q.enqueue('d');
q.enqueue('a');
q.enqueue('y');
q.enqueue('i');
q.enqueue('s');
q.dequeue();
q.enqueue('h');
q.enqueue('a');
```

Iterators

In C++, iterators provide an interface for client code access to data in a way that abstracts away the internals of the data structure.

An instance of an iterator is a current location in a pass through the data structure:

<table>
<thead>
<tr>
<th>Type</th>
<th>Cur. Location</th>
<th>Current Data</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linked List</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Array</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hypercube</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The iterator minimally implements three member functions:

- `operator*`, Returns the current data
- `operator++`, Advance to the next data
- `operator!=`, Determines if the iterator is at a different location

Implementing an Iterator

A class that implements an iterator must have two pieces:

1. [Implementing Class]: Must implement:
   - 
   - 

2. [Implementing Class’ Iterator]:
   A separate class (usually an internal class) that extends `std::iterator` and implements an iterator. This requires:
   - 
   - 

Locations of ::begin and ::end iterators:

<table>
<thead>
<tr>
<th>Type</th>
<th>::begin()</th>
<th>::end()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linked List</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Array</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CS 225 – Things To Be Doing:

1. lab_memory due Sunday
2. mp_list extra credit part1 due Monday
3. Daily POTDs