Xavier Kuehn

1. What are the iterator invalidation rules of std::list?
   - Insertion: iterators and references remain unaffected.
   - Erasure: only the iterators of the erased object become invalidated.

2. What are the iterator invalidation rules for STL's vector class?
   - Insertion: All iterators after the point of insertion are invalidated. If reallocation is necessary, then all iterators are invalidated.
   - Erasure: All iterators at the point of erasure and after are invalidated.

3. What are the features of a random-access iterator? Present the name of two STL data structures that offer random access iterators.
   - The random access iterator can read and write and allows for random access to any element within the data structure; a random access iterator p can access the nth element of the structure by p[n], or moves the iterator n steps in front of the current item.
   - Vector, Deque

4. Write the pseudocode of an algorithm that evaluates a fully parenthesized mathematical expression using stack data structure. (calculator).

```
- We will use two stacks, one for the numbers and one for the
  operators
- Parse through an expression, when a number is reached, push it
  to the number stack. When an operator is reached, push it to the
  operator stack. Ignore left parentheses and white spaces.
- Once a right parenthesis is read, combine the two top numbers in
  the number stack via the operation at the top of the operator
  stack. Push the calculation result to the numbers stack (after
  popping the operands) and pop the operator that was used.
- Continue until the end of the expression is reached. The
  operator stack should be empty while the number stack should
  have only the final result.
```

5. Please write the implementation of a template const forward iterator for the node class.

```
template <class Item>
class const_node_iterator : public std::iterator<std::forward_iterator_tag, Item> {
	public:
	const_node_iterator(const node<Item>* (initial = NULL) {current = initial;}
	const Item& operator *() const {return current->data();}
	const_node_iterator& operator ++() {
		current = current->link();
		return *this;
	}
```

```
        const_node_iterator operator ++(int){
              const const_node_iterator original(current);
              current = current->link( );
              return original;
        }

        bool operator ==(const const_node_iterator other) const
        { return current == other.current;}

        bool operator !=(const const_node_iterator other) const
        { return current != other.current; }

        private:
        const node<Item>* current;
};
```

6. Complete the implementation of the bag iterator.

```
template <class Item>
class bag_iterator {
        private:
        size_type capacity;
        size_type used;
        size_type current;
        Item* data;

        public:
        bag_iterator(size_t cap, size_t use, size_t pos, Item* el){
              current = pos;
              capacity = cap;
              used = count;
              data = el;
        }

        Item& operator *() const {return *data;}
        bag_iterator& operator ++() {
              ++current;
              return *this;
        }
        bag_iterator operator ++(int) {
              bag_iterator og(capacity, used, current, data);
              ++current;
              return og;
        }
        bool operator ==(const bag_iterator other) const
        { return current == other.current;}
        bool operator !=(const bag_iterator other) const
        { return current != other.current;}
};
```

7. For the queue class given in Appendix 1 (cf. end of this assignment), implement the copy constructor.

```
template <class Item>
queue(const queue& source){
      first = source.first;
      last = source.last;
      count = source.count;
      capacity = source.capacity;
      data = new Item[capacity];
      std::copy(data, data + count, source.data);
}
```

8. For the queue class given in Appendix 1 (cf. end of this assignment), implement the following function, which increases the size of the dynamic array used to store items.

```
template <class Item>
void queue<Item>::reserve (size_type new_capacity) {
      value_type* larger_array;

      // check capacity
      if (new_capacity == capacity) return;
      if (new_capacity < count) new_capacity = count;

      larger_array = new value_type[new_capacity];
      size_t cnt = count;
      for (int i = 0; i < cnt; ++i) {
            larger_array[i] = front();
            pop();
      }
      count = cnt;
      front = 0;
      last = count - 1;
      capacity = new_capacity;
}
```

9. For the deque class given in Appendix 2 (cf. end of this assignment), implement the following constructor.

```
template <class Item>
deque<Item>::deque(int init_bp_array_size, int init_block_size){
      bp_array_size = init_bp_array_size;
      block_size = init_block_size;

      block_pointers = new value_type* [bp_array_size];
      for (size_type index = 0; index < bp_array_size; ++index){
      block_pointers[index] = NULL;
      }

      block_pointers_end = block_pointers + (bp_array_size - 1);
      first_bp = last_bp = NULL;
      front_ptr = back_ptr = NULL;
}
```

10. For the deque class given in Appendix 2 (cf. end of this assignment), write the full implementation of the following function.

```
void deque <Item>::pop_front(){
// Precondition: There is at least one entry in the deque
// Postcondition: Removes an item from the front of the deque
        assert(!isEmpty());
        // case 1: only 1 item in deque
        if (front_ptr == back_ptr) clear();

        // case 2: front item is last element in block
        else if (front_ptr == (*first_bp + block_size - 1)) {
                delete[] first_bp;
                *first_bp = NULL:
                --first_bp;
                front_ptr = *first_bp;
        }

        // case 3: front item is somewhere in middle or front of block
        else --front_ptr;
}
```