COEN 79

Object-Oriented Programming and Advanced Data Structures

Assignment #5 - Due: November 15, 2018 – 11:59PM

Name: Jordan Murtiff **Date:** 11-21-18

- Number of questions: 10
- Points per question: 0.3
- Total: 3 points

---

1. What are the *iterator invalidation rules* for a data structure that stores items in a *linked list*?

There are two iterator invalidation rules for a data structure that stores items in a linked list:

1. When you insert into a linked list, there is no iterator invalidation.
2. If you delete the node that an iterator is pointing to, then the iterator is invalid.

2. What are *the iterator invalidation rules* for STL's `vector` class?

There are five iterator invalidation rules for the STL's vector class:

1. If we reallocate the array, all iterators are invalid.
2. If an element is inserted into the vector, and the array is not reallocated, then any insertion at or before where the iterator is pointing to makes the iterator invalid.
3. If an element is inserted into the vector, and the array is not reallocated, then any insertion after where the iterator is pointing to makes the iterator still valid.
4. If you delete an element contained within a vector object at or before where the iterator is pointing to, then the iterator is invalid.
5. If you delete an element contained within a vector object after the iterator, then the iterator is still valid.

3. What are the features of a *random access iterator*?
Present the name of two STL data structures that provide random access iterators.

The features of a random access iterator are:
1. Has all the abilities of bidirectional iterators (so the iterator can move both forward and backward through a data structure).
2. Can instanly access any item within a data structure in O(1) time.
3. Can both read and write to any element within a data structure.

Two STL data structures that provide random access iterators are the vector data structure (dynammicaly allocated arrays) and the deque data structure (double-ended queue).

4. Write the *pseudo-code* of an algorithm for evaluating a *fully parenthesized mathematical expression* using *stack* data structure. (calculator).

Using two stacks we can evaluate a fully parenthesized mathematical expression.
One stack will be a stack of characters while the other will be a stack of double numbers.

```
while (the character being read is not the NULL terminator character)
{
    Read one character from the equation.
    if (the character is a number)
        Push the number onto the numbers stack.
    else if (the character is an operation character)
        Push the operation character into the operations stack.
    else if(the character is a left parenthesis or a blank space)
        Ignore the character and do not do anything.
    else if(the character is a right parenthesis)
        Pop the top two numbers from the top of the numbers stack and pop the operation
        on the top of the operation stack.
        Combine the two numbers using the given operation, and push the resulting number back
        on to the number stack.
}
Return the number on top of the number stack, that is the final answer.
```

5. The node class is defined as follows:

```
1.  template <class Item>
2.  class node {
3.  public:
4.      // TYPEDEF
5.      typedef Item value_type;
6.
7.      // CONSTRUCTOR
8.      node( const Item& init_data = Item(), node* init_link = NULL ) {
9.              data_field = init_data;
10.             link_field = init_link;
11.         }
12.
13.     // MODIFICATION MEMBER FUNCTIONS
14.     Item& data() {  return data_field; }
15.     node* link() {  return link_field; }
16.     void set_data(const Item& new_data) { data_field = new_data;  }
17.     void set_link(node* new_link) {  link_field = new_link;  }
18.
19.     // CONST MEMBER FUNCTIONS
20.     const Item& data() const {  return data_field;  }
21.     const node* link() const {  return link_field;  }
22.
23. private:
24.     Item data_field;
25.     node* link_field;
26. };
```

Write the implementation of a *const forward iterator* for this class. Use *inline* functions in your implementation. The iterator is a *template class*.

Answer:

```
template <class Item>
class const_node_iterator:
    public std::iterator<std::forward_iterator_tag, const 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_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. The bag class is defined as follows:

```
template < class Item >
class bag {
public:
    // TYPEDEFS and MEMBER CONSTANTS
    typedef Item value_type;
    typedef std::size_t size_type;

    static const size_type DEFAULT_CAPACITY = 30;

    typedef bag_iterator < Item > iterator;

    bag(size_type initial_capacity = DEFAULT_CAPACITY);
    bag(const bag& source);
    ~bag();

    // MODIFICATION MEMBER FUNCTIONS
    // ...

    iterator begin();
    iterator end();

private:
    Item* data;          // Pointer to partially filled dynamic array
    size_type used;      // How much of array is being used
    size_type capacity;  // Current capacity of the bag
};
```

- This class implements the following functions to create iterators:

```
template <class Item>
typename bag <Item>::iterator bag<Item>::begin() {
    return iterator(capacity, used, 0, data);
}

template <class Item>
typename bag<Item>::iterator bag<Item>::end() {
    return iterator(capacity, used, used, data);
}
```

- Complete the implementation of the following iterator:

```
template < class Item >
class bag_iterator: public std::iterator < std::forward_iterator_tag, Item >
{
public:
    typedef std::size_t size_type;
    bag_iterator(size_type capacity, size_type used, size_type current, Item* data) {
        this -> capacity = capacity;
```

```
8.          this -> current = current;
9.          this -> used = used;
10.         this -> data = data;
11.     }
12.
```

```
Item& operator*() const
{
    return data[current];
}

bag_iterator& operator ++()
{
    current++;
    data++;
    return *this;
}

bag_iterator operator ++(int)
{
    bag_iterator original(capacity, used, current, data);
    current++;
    data++;
    return original;
}

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

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

```
13. private:
14.     size_type capacity;
15.     size_type used;
16.     size_type current;
17.     Item* data;
18. };
```

7. For the `queue` class given in Appendix 1 (cf. end of this assignment), implement the *copy constructor*.
   Note that the class uses a dynamic array. Also please note that you should not use the `copy` function (copy only the valid entries of one array to the new array).

```
template <class Item>
queue<Item>::queue (const queue <Item>& source)
{
    data = new value_type[source.capacity];  // Allocate an array

    capacity = source.capacity;
```

```
value_type* runner = &data[source.first];
value_type* end = &data[source.last];
size_type i = source.first;
while(runner != end+1)
{
    data[i] = source.data[i];
    runner++;
    i = (i+1) % capacity;
}
count = source.count;
first = source.first;
last = source.last;
}
```

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. Please note that you should not use the `copy` function (copy only the valid entries of one array to the new array).

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

        if (new_capacity == capacity) return;

        if (new_capacity < count)
            new_capacity = count;
```

```
larger_array = new value_type[new_capacity];
value_type* runner = &data[first];
value_type* end = &data[last];
size_type i = first;
size_type j  = 0;
while(runner != end+1)
{
    larger_array[j] = data[i];
    runner++;
    i = (i+1) % capacity;
    j++;
}
delete [] data;
data = larger_array;
capacity = new_capacity;
first = 0;
last = count-1;
}
```

9. For the `deque` class given in Appendix 2 (cf. end of this assignment), implement the following constructor. The constructor allocates an array of block pointers and initializes all of its entries with NULL. The initial size of the array is `init_bp_array_size`.

```
1.  template < class Item >
2.  deque<Item>::deque( int init_bp_array_size, int init_block_size )
3.  {
4.          bp_array_size = init_bp_array_size;
5.          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.

```
1.  template <class Item>
2.  void deque <Item>::pop_front()
3.  // Precondition: There is at least one entry in the deque
4.  // Postcondition: Removes an item from the front of the deque
5.  {
6.      assert(!isEmpty());
7.      if (back_ptr == front_ptr)
8.      {
            clear();
9.      }
10.
11.     // The front element is the last element of the first block
12.     else if (front_ptr == ((*first_bp) + block_size - 1))
13.     {
14.
            delete [] *first_bp;
            *first_bp = NULL;
            ++first_bp;
            front_ptr = *first_bp
15.     }
16.
17.     else
18.     {
            ++front_ptr;
19.     }
20. }
```

Appendix 1: queue class

```
template < class Item >
class queue {
public:
    // TYPEDEFS and MEMBER CONSTANTS
    typedef std::size_t size_type;
    typedef Item value_type;

    static const size_type CAPACITY = 30;

    // CONSTRUCTOR and DESTRUCTOR
    queue(size_type initial_capacity = CAPACITY);
    queue(const queue& source);
    ~queue();

    // MODIFICATION MEMBER FUNCTIONS
    Item& front();
    void pop();
    void push(const Item & entry);
    void reserve(size_type new_capacity);

    // CONSTANT MEMBER FUNCTIONS
    bool empty() const {  return (count == 0);  }
    const Item & front() const;
    size_type size() const { return count;  }

private:
    Item* data;         // Circular array
    size_type first;     // Index of item at front of the queue
    size_type last;      // Index of item at rear of the queue
    size_type count;     // Total number of items in the queue
    size_type capacity;  // HELPER MEMBER FUNCTION

    size_type next_index(size_type i) const  { return (i + 1) % capacity; }
};
```

Appendix 2: deque class

```
template < class Item >
class deque {
public:
    // TYPEDEF
    static const size_t BLOCK_SIZE = 5; // Number of data items per block

    // Number of entries in the block of array pointers. The minimum acceptable value is 2
    static const size_t BLOCKPOINTER_ARRAY_SIZE = 5;

    typedef std::size_t size_type;
    typedef Item value_type;

    deque(int init_bp_array_size = BLOCKPOINTER_ARRAY_SIZE,
                              int initi_block_size = BLOCK_SIZE);

    deque(const deque & source);
    ~deque();

    // CONST MEMBER FUNCTIONS
    bool isEmpty();
    value_type front();
    value_type back();

    // MODIFICATION MEMBER FUNCTIONS
    void operator = (const deque & source);
    void clear();
    void reserve();
    void push_front(const value_type & data);
    void push_back(const value_type & data);
    void pop_back();
    void pop_front();

private:
    // A pointer to the dynamic array of block pointers
    value_type** block_pointers;

    // A pointer to the final entry in the array of block pointers
    value_type** block_pointers_end;

    // A pointer to the first block pointer that's now being used
    value_type** first_bp;

    // A pointer to the last block pointer that's now being used
    value_type** last_bp;

    value_type* front_ptr; // A pointer to the front element of the whole deque
    value_type* back_ptr; // A pointer to the back element of the whole deque

    size_type bp_array_size; // Number of entries in the array of block pointers
    size_type block_size; // Number of entries in each block of items
};
```