COEN 79

Object-Oriented Programming and Advanced Data Structures

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

Name: **Date:**

- 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*?

Answer:
As we do not need to shift elements in memory when new items are added or removed, iterators are not invalidated.
Only the iterator to the removed element is invalid.

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

Answer:
The STL's `vector` class is similar to a bag class with dynamic array. Therefore, the iterator invalidation rules are as follows.

Insertion:

- Case 1: If a new array is not allocated, then all the iterators at or after the point of insertion are invalid because of shifting the elements.
- Case 2: If a new array is allocated, then all the iterators are invalid.

Removal: All the iterators at or after the point of removal are invalid because of shifting the elements.

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

A random access iterator supports increment (++), decrement (--), dereferencing (*) and random access ([]) operators.
The term random access refers to the ability to quickly access any randomly selected location in a data structure. A random access iterator `p` can use the notation `p[n]` (to provide access to the item that is `n` steps in front of the current item. Therefore, distant elements can be accessed directly by applying an offset value to an iterator without iterating through all the elements in between.
Produced by: ordinary pointers; vector; deque

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

```
main() {
    while (there is an input character)
    {
        if (the input character is a digit || it is a decimal point) {
            push the number into the “numbers” stack;
        }
        else if (the input character is “+”, “-”, “*” or “/”) {
            push the number into the “operators” stack;
        }
        else if ( the input character is a “)” ) {
              evaluate the operation on top of the “operations” stack on the two numbers
               on top of the “numbers” stack;
        }
        else ignore the input;
    }
}
```

5. The node class is defined as follows:

```
template <class Item>
class node {
public:
    // TYPEDEF
    typedef Item value_type;

    // CONSTRUCTOR
    node( const Item& init_data = Item(), node* init_link = NULL ) {
            data_field = init_data;
            link_field = init_link;
        }

    // MODIFICATION MEMBER FUNCTIONS
    Item& data() {  return data_field; }
    node* link() {  return link_field; }
    void set_data(const Item& new_data) { data_field = new_data;  }
    void set_link(node* new_link) {  link_field = new_link;  }

    // CONST MEMBER FUNCTIONS
    const Item& data() const {  return data_field;  }
    const node* link() const {  return link_field;  }

private:
    Item data_field;
    node* link_field;
};
```

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

Answer:

```
1.  template <class Item>
2.  class const_node_iterator: public std::iterator <std::forward_iterator_tag, const Item>
3.  {
4.  public:
5.      const_node_iterator(const node <Item>* initial = NULL) {
6.              current = initial;  }
7.
8.      const Item& operator* () const {  return current -> data();  }
9.
10.     const_node_iterator & operator++() // Prefix ++
11.     {
12.         current = current -> link();
13.         return *this;
14.     }
15.
16.     const_node_iterator operator++(int) // Postfix ++
17.     {
18.         const_node_iterator original(current);
19.         current = current -> link();
20.         return original;
21.     }
22.
23.     bool operator == (const const_node_iterator other) const {
24.         return current == other.current;
25.     }
26.
27.     bool operator != (const const_node_iterator other) const {
28.         return current != other.current;
29.     }
30.
31. private:
32.     const node <Item>* current;
33. };
```

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;
        this -> current = current;
```

```
9.          this -> used = used;
10.         this -> data = data;
11.     }
12.
13.     Item& operator* () const {
14.         return data[current];
15.     }
16.
17.     bag_iterator& operator++()   // Prefix ++
18.     {
19.         if (current != used) ++current;
20.         return *this;
21.     }
22.
23.     bag_iterator operator++(int) // Postfix ++
24.     {
25.         bag_iterator original(capacity, used, current, data);
26.         if (current != used) ++current;
27.         return original;
28.     }
29.
30.     bool operator == (const bag_iterator other) const {
31.         return (capacity == other.capacity && current == other.current
           && used == other.used && data == other.data);
32.     }
33.
34.     bool operator != (const bag_iterator other) const {
35.         return (capacity != other.capacity || current != other.current
           || used != other.used || data != other.data);
36.     }
37.
38. private:
39.     size_type capacity;
40.     size_type used;
41.     size_type current;
42.     Item* data;
43. };
```

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).

```
1.  template <class Item>
2.  queue<Item>::queue (const queue <Item>& source)
3.  {
4.      data = new value_type[source.capacity];  // Allocate an array
5.
6.      capacity = source.capacity;
7.
8.      count = source.count;
9.      first = source.first;
10.     last  = source.last;
11.
12.     if (source.count != 0)  // Copy the elements from source to the newly created array
13.     {
14.         size_type tmpCount = count;
15.         size_type tmpCursor = first;
16.
17.         while (tmpCount > 0) {
18.             data[tmpCursor] = source.data[tmpCursor];
19.             tmpCursor = (tmpCursor + 1) % capacity;
20.             --tmpCount;
21.         }
22.     }
23. }
```

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).

```
1.  template<class Item>
2.  void queue<Item>::reserve (size_type new_capacity)
3.  {
4.          value_type* larger_array;
5.
6.          if (new_capacity == capacity) return;
7.
8.          if (new_capacity < count)
9.              new_capacity = count;
10.
24.         larger_array = new value_type[new_capacity];  // Allocate a new array
11.
12.
13.         if (count == 0) {
14.             first = 0;
15.             last = new_capacity - 1;
16.         }
25.         else  // Copy the elements from old array to the new array
17.         {
18.             size_type tmpCount = count;
19.             size_type new_last = new_capacity - 1;
20.
21.             while (tmpCount > 0) {
22.                 new_last = (new_last + 1) % new_capacity;
23.                 larger_array[new_last] = data[first];
24.                 first = (first + 1) % capacity;
25.                 --tmpCount;
26.             }
27.
28.             first = 0;
29.             last = new_last;
30.         }
31.
32.         capacity = new_capacity;
26.         delete[] data;  // Delete the old array
33.         data = larger_array;
34. }
```

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`.

```
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;

        //set a pointer the start of the array of block pointers
        block_pointers = new value_type* [bp_array_size];

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

        //set a pointer to the end of the array of block pointers
        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.

```
template <class Item>
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());
    if (back_ptr  ==  front_ptr)
    {
        // This is the only element, clear the deque
        for (size_type index = 0; index < bp_array_size; ++index)
        {
            delete[] block_pointers[index];
            block_pointers[index] = NULL;
        }

        first_bp = last_bp = NULL;
        front_ptr = back_ptr = NULL;
    }

    // The front element is the last element of the first block
    else if (front_ptr == (( *first_bp)  +  block_size - 1))
    {
        delete[] *first_bp;
        *first_bp = NULL;
        ++first_bp;
        front_ptr = *first_bp;
    }

    else
    {
       ++front_ptr;
    }
}
```

Appendix 1: queue class declaration:

```
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 declaration:

```
1.  template < class Item >
2.  class deque {
3.  public:
4.      // TYPEDEF
5.      static const size_t BLOCK_SIZE = 5; // Number of data items per block
6.
7.      // Number of entries in the block of array pointers. The minimum acceptable value is 2
8.      static const size_t BLOCKPOINTER_ARRAY_SIZE = 5;
9.
10.     typedef std::size_t size_type;
11.     typedef Item value_type;
12.
13.     deque(int init_bp_array_size = BLOCKPOINTER_ARRAY_SIZE,
14.                               int initi_block_size = BLOCK_SIZE);
15.
16.     deque(const deque & source);
17.     ~deque();
18.
19.     // CONST MEMBER FUNCTIONS
20.     bool isEmpty();
21.     value_type front();
22.     value_type back();
23.
24.     // MODIFICATION MEMBER FUNCTIONS
25.     void operator = (const deque & source);
26.     void clear();
27.     void reserve();
28.     void push_front(const value_type & data);
29.     void push_back(const value_type & data);
30.     void pop_back();
31.     void pop_front();
32.
33. private:
34.     // A pointer to the dynamic array of block pointers
35.     value_type** block_pointers;
36.
37.     // A pointer to the final entry in the array of block pointers
38.     value_type** block_pointers_end;
39.
40.     // A pointer to the first block pointer that's now being used
41.     value_type** first_bp;
42.
43.     // A pointer to the last block pointer that's now being used
44.     value_type** last_bp;
45.
46.     value_type* front_ptr; // A pointer to the front element of the whole deque
47.     value_type* back_ptr; // A pointer to the back element of the whole deque
48.
49.     size_type bp_array_size; // Number of entries in the array of block pointers
50.     size_type block_size; // Number of entries in each block of items
51. };
```