COEN 79

Object-Oriented Programming and Advanced Data Structures

Assignment #5 -

Name: **Date:**

- Number of questions: 6
- Points per question: 4

1. Write the *pseudo-code* of an algorithm which evaluates a *fully parenthesized mathematical expression* using *stack* data structure. (calculator).

2. 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);
}
```

- Please 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 -> used = used;
        this -> current = current;
        this -> data = data;
    }

    Item& operator* () const {


    }

    bag_iterator& operator++()   // Prefix ++
    {

```

```
    }

    bag_iterator operator++(int) // Postfix ++
    {


    }

    bool operator == (const bag_iterator other) const
    {


    }

    bool operator != (const bag_iterator other) const
    {


    }

private:
    size_type capacity;
    size_type used;
    size_type current;
    Item* data;
};
```

3. 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)
```

4. 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;
```

5. 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;
```

6. 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());
```

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. };
```