Lab Assignment #3 **[Weight: ~6% of the Course Grade]**

Topic: Heap ADT and Binary Search Tree ADT Applications

- **For this assignment, you must work in pairs or — exceptionally — in teams of three.**
- **Include the name and ID for each group member in your files.**
- **For C/C++ implementation, you must separate class header and class implementation. You may not use other languages for this assignment to ensure adequate preparation for the final exam.**
- **Please submit your completed assignment before the dropbox closes on LEARN.**

Part1. Task Organizer using Heap ADT

We are going to design a task organizer for management of daily personal tasks, and also for eventual sharing of tasks among different users.

For Part1, we are going to use (Max) Heap ADT to model and implement this prototype. Submit two files: **`lab3_ priority_queue.hpp` and `lab3_ priority_queue.cpp`**. Do not modify the signatures for any of the functions already implemented or listed below.

Step0. [see `lab3_ priority_queue.hpp`]

Start by familiarizing yourself with the code in the provided header file.

Each task item will be recorded as an instance of **`PriorityQueue::TaskItem`** class. Each record will include item priority stored as an integer and item description stored as a string. **`PriorityQueue`** class will be used as a container to handle **`TaskItem`** objects. **`PriorityQueue`** will include relevant member attributes and methods. The highest priority item will always be stored at the top.

Step1. [see `lab3_ priority_queue.cpp`]

Implement all of the methods for the class **`PriorityQueue`** that are listed below.

`class PriorityQueue {…}` [header already defined in `lab3_ priority_queue.hpp`;
provide your definitions in `lab3_ priority_queue.cpp`]

```
// CONSTRUCTOR AND DESTRUCTOR

// PURPOSE: Parametric constructor
// initializes heap to an array of (n_capacity + 1) elements
PriorityQueue(unsigned int n_capacity);

// PURPOSE: Explicit destructor of the class PriorityQueue
~PriorityQueue();
```

```
// ACCESSORS

// PURPOSE: Returns the number of elements in the priority queue
unsigned int get_size() const;

// PURPOSE: Returns true if the priority queue is empty; false, otherwise
bool empty() const;

// PURPOSE: Returns true if the priority queue is full; false, otherwise
bool full() const;

// PURPOSE: Prints the contents of the priority queue; format not specified
void print() const;

// PURPOSE: Returns the max element of the priority queue without removing it
// if the priority queue is empty, it returns (-1, "N/A")
TaskItem max() const;


// MUTATORS

// PURPOSE: Inserts the given value into the priority queue
// re-arranges the elements back into a heap
// returns true if successful and false otherwise
// priority queue does not change in capacity
bool enqueue(TaskItem val);

// PURPOSE: Removes the top element with the maximum priority
// re-arranges the remaining elements back into a heap
// returns true if successful and false otherwise
// priority queue does not change in capacity
bool dequeue();
```

Step2. [see `lab3_tests.cpp`]

Ensure that your code passes all of the test case methods in **`PriorityQueueTest`** listed below.

`class PriorityQueueTest {…}` [implemented in `lab3_tests.cpp`]

```
// TEST CASES
// PURPOSE: Tests if the new queue is valid
bool test1();

// PURPOSE: Tests enqueue of one item and then dequeue of that item
bool test2();

// PURPOSE: Tests enqueue too many
bool test3();

// PURPOSE: Tests enqueue too many then dequeue too many
bool test4();
```

Part2. Task Organizer using Binary Search Tree ADT

For Part2, we are going to use Binary Search ADT to model and implement this prototype. Submit two files: **`lab3_binary_search_tree.hpp` and `lab3_binary_search_tree.cpp`**. Do not modify the signatures for any of the functions already implemented or listed below.

Step0. [see `lab3_binary_search_tree.hpp`]

Start by familiarizing yourself with the code in the provided header file.

Each task item will be recorded as an instance of a slightly modified **`BinarySearchTree::TaskItem`** class. Each record will include item priority stored as an integer and item description stored as a string along with corresponding left and right pointers. **`BinarySearchTree`** class will be used as a container to handle **`TaskItem`** objects. **`BinarySearchTree`** will also include relevant member attributes and methods.

Step1. [see `lab3_binary_search_tree.cpp`]

Implement all of the methods for the class **`BinarySearchTree`** that are listed below.

`class BinarySearchTree {…}` [header already defined in `lab3_binary_search_tree.hpp`; provide your definitions in `lab3_binary_search_tree.cpp`]

```
// CONSTRUCTOR AND DESTRUCTOR

// PURPOSE: Default/empty constructor
 BinarySearchTree();

// PURPOSE: Explicit destructor of the class BinarySearchTree
~BinarySearchTree();


// ACCESSORS

// PURPOSE: Returns the number of nodes in the tree
unsigned int get_size() const;

// PURPOSE: Returns the maximum value of a node in the tree
// if the tree is empty, it returns (-1, "N/A")
TaskItem max() const;

// PURPOSE: Returns the minimum value of a node in the tree
// if the tree is empty, it returns (-1, "N/A")
TaskItem min() const;

// PURPOSE: Returns the tree height
unsigned int height() const;

// PURPOSE: Prints the contents of the tree; format not specified
void print() const;
```

```
// PURPOSE: Returns true if a node with the value val exists in the tree
// otherwise, returns false
bool exists(TaskItem val) const;

// MUTATORS

// PURPOSE: Inserts the value val into the tree if it is unique
// returns true if successful; returns false if val already exists
bool insert(TaskItem val);

// PURPOSE: Removes the node with the value val from the tree
// returns true if successful; returns false otherwise
bool remove(TaskItem val);
```

Step2. [see `lab3_tests.cpp`]

Ensure that your code passes all of the test case methods in **`BinarySearchTreeTest`** listed below.

`class BinarySearchTreeTest{…}` [implemented in `lab3_tests.cpp`]

```
// TEST CASES
// PURPOSE: Tests if the new tree is valid
bool test1();

// PURPOSE: Tests a tree with one node
bool test2();

// PURPOSE: Tests insert, remove, and size on linear list formation with three elements
bool test3();

// PURPOSE: Tests removal of a node with one child
bool test4();

// PURPOSE: Tests insert of multiple elements and remove till nothing remains
bool test5();

// PURPOSE: Tests removal of root node when both children of root have two children
bool test6();

// PURPOSE: Tests depth with many inserts and some removes
bool test7();

// PURPOSE: Tests lots of inserts and removes
bool test8();
```