SYDE 223 – Assignment #3
Priority Queue and Binary Search Tree

Overview
Join up with your Assignment #3 team as assigned on LEARN. Download the six files made available with the course assignment (`a3_priority_queue.hpp`, `a3_priority_queue.cpp`, `a3_binary_search_tree.hpp`, `a3_binary_search_tree.cpp`, `a3_tests.hpp`, `a3_main.cpp`), add them to a new project in Dev-C++, or a matching development environment, and ensure that the project compiles and runs.

As reference, please review Chapter 7 of the course handbook.

Part1: Priority Queue Implementation [25 marks]
Let us implement the Maximum Priority Queue ADT.

- The header file **`a3_priority_queue.hpp`**, which is included below, provides the structure of the **`PriorityQueue`** class with declarations of member functions. Do not modify the signatures of any of these functions.

- Implement all of the public member functions listed in **`a3_priority_queue.hpp`** and place them into **`a3_priority_queue.cpp`**. Both files can be downloaded from LEARN along with other A3-related files.

- Note that this class defines a priority queue, implemented as a heap. A higher value indicates a higher priority (e.g., an element with value 5 has a higher priority than an element with value 1).

- The elements in the heap are represented using a sequential array as explained in the textbook. For simplicity, this array does not grow or shrink. Note that if you are given a capacity of 16, this indicates the heap should store 16 elements. Your array should be of size 17 since we do not use index 0 in the sequential representation of a heap.

The details of the header file **`a3_priority_queue.hpp`** are as follows:

- **`DataType`** defines the kind of data that the queue will contain. Being public, it can be accessed directly as **`PriorityQueue::DataType`**.

- Member variables:
  1. **`heap_`**: Sequential representation of the priority queue.
  2. **`capacity_`**: Total number of elements that the priority queue can store.
  3. **`size_`**: Current number of elements in the priority queue.

- Constructor and Destructor:
    1. **`PriorityQueue(unsigned int capacity)`**: Constructor initializes **`heap_`** to an array of **`(capacity_ + 1)`** size, so that there are at most **`capacity_`** elements in the priority queue.
    2. **`~PriorityQueue()`**: Destructor of the class **`PriorityQueue`**. It deallocates the memory space allocated for the priority queue.

- Constant member functions:
    1. **`int size() const:`** Returns the number of elements in the priority queue.
    2. **`bool empty() const:`** Returns **`true`** if the priority queue is empty, and **`false`** otherwise.
    3. **`bool full() const:`** Returns **`true`** if the priority queue is full, and **`false`** otherwise.
    4. **`void print() const:`** Prints the contents of the priority queue.
    5. **`DataItem max() const:`** Returns the max element of the priority queue, but does not remove it.

- Non-constant member functions:
    1. **`bool enqueue(DataItem value):`** Inserts **`value`** into the priority queue. Returns **`true`** if successful, and **`false`** otherwise. Assume no duplicate entries will be entered.
    2. **`bool dequeue():`**Removes the top element with the maximum value (priority) and rearranges the resulting heap. Returns **`true`** if successful, and **`false`** otherwise.

```
#ifndef A3_PRIORITY_QUEUE_HPP
#define A3_PRIORITY_QUEUE_HPP

class PriorityQueue {
public:
      // Can be seen externally as PriorityQueue::DataType
      typedef int DataType;

private:
      friend class PriorityQueueTest;

      // Sequential representation of the priority queue.
      DataType* heap_;

      // Total number of elements that the priority queue can store.
      unsigned int capacity_;

      // Current number of elements in the priority queue.
      unsigned int size_;

      // Override copy constructor and assignment operator in private so we can't
      // use them.
      PriorityQueue(const PriorityQueue& other) {}
      PriorityQueue& operator=(const PriorityQueue& other) {}
```

```
public:
      // Constructor initializes heap_ to an array of (capacity_ + 1) size, so
      // that there are at most capacity_ elements in the priority queue.
      PriorityQueue(unsigned int capacity);
      // Destructor of the class PriorityQueue. It deallocates the memory space
      // allocated for the priority queue.
      ~PriorityQueue();

      // Returns the number of elements in the priority queue.
      unsigned int size() const;

      // Returns true if the priority queue is empty, and false otherwise.
      bool empty() const;
      // Returns true if the priority queue is full, and false otherwise.
      bool full() const;
      // Prints the contents of the priority queue.
      void print() const;
      // Returns the max element of the priority queue, but does not remove it.
      DataType max() const;


      // Inserts value into the priority queue. Returns true if successful, and
      // false otherwise. Assume no duplicate entries will be entered.
      bool enqueue(DataType val);
      // Removes the top element with the maximum value (priority) and rearranges
      // the resulting heap. Returns true if successful, and false otherwise.
      bool dequeue();
};
#endif
```

Part2: Binary Search Tree Implementation [25 marks]

Let us implement the Binary Search Tree ADT.

- The header file **`a3_binary_search_tree.hpp`**, which is included below, provides the structure of the **`BinarySearchTree`** class with declarations of member functions. Do not modify the signatures of any of these functions.

- Implement all of the public member functions listed in **`a3_binary_search_tree.hpp`** and place them into **`a3_binary_search_tree.cpp`**. Both files can be downloaded from LEARN along with other A3-related files.

- Note that this tree does not have to be balanced.

The details of the header file **`a3_binary_search_tree.hpp`** are as follows:

- **`DataType`** defines the kind of data that the tree will contain. Being public, it can be accessed directly as **`BinarySearchTree::DataType`**.

- Member variables:
    1. **`Node`**: This is a structure declaration. **`Node`** contains following member variables:
        - **`Node(DataType newval):`** Sets the left and right children to **`NULL`**, and initializes **`val`**.
        - **`val`**: Value of the node.
        - **`left`**: Pointer to the left node.

        - **right**: Pointer to the right node.
    2. **root_**: Pointer to the root node of the tree.
    3. **size_**: Number of nodes in the tree.

- Constructor and Destructor:
    1. **BinarySearchTree()**: Default constructor to initialize the root.
    2. **~BinarySearchTree()**: Destructor of the class **BinarySearchTree**. It deallocates the memory space allocated for the binary search tree.

- Constant member functions:
    1. **int size() const:** Returns the number of nodes in the tree.
    2. **DataItem max() const:** Returns the maximum value of a node in the tree. You can assume that this function will never be called on an empty tree.
    3. **DataItem min() const:** Returns the minimum value of a node in the tree. You can assume that this function will never be called on an empty tree.
    4. **unsigned int depth() const:** Returns the maximum depth of the tree. A tree with only the root node has a depth of 0. You can assume that this function will never be called on an empty tree.
    5. **void print() const:** You can print the tree in whatever order you prefer. However, methods such as in-order or level-order traversal could be the most useful for debugging.
    6. **bool exists(DataType val) const:** Returns **true** if a node with the value **val** exists in the tree; otherwise, it returns **false**.

- Non-constant member functions:
    1. **bool insert(DataType val):** Inserts the value **val** into the tree. Returns **false** if **val** already exists in the tree, and **true** otherwise.
    2. **bool remove(DataType val):** Removes the node with the value **val** from the tree. Returns **true** if successful, and **false** otherwise.

```
#ifndef A3_BINARY_SEARCH_TREE_HPP
#define A3_BINARY_SEARCH_TREE_HPP

class BinarySearchTree {
public:
      // Can be seen externally as BinarySearchTree::DataType
      typedef int DataType;

      struct Node {
            // Sets the left and right children to NULL, and initializes val.
            Node(DataType newval);

            DataType val;  // Value of the node.
            Node* left;    // Pointer to the left node.
            Node* right;   // Pointer to the right node.
      };
```

```
private:
	friend class BinarySearchTreeTest;

	// Optional function that recursively gets the maximum depth for a given node.
	int getNodeDepth(Node* n) const;

	// Pointer to the root node of the tree.
	Node* root_;
	// Number of nodes in the tree.
	unsigned int size_;

	// Sets copy constructor and assignment operator to private.
	BinarySearchTree(const BinarySearchTree& other) {}
	BinarySearchTree& operator=(const BinarySearchTree& other) {}

public:
	// Default constructor to initialize the root.
	BinarySearchTree();
	// Destructor of the class BinarySearchTree. It deallocates the memory
	// space allocated for the binary search tree.
	~BinarySearchTree();
	// Returns the number of nodes in the tree.
	unsigned int size() const;
	// Returns the maximum value of a node in the tree. You can assume that
	// this function will never be called on an empty tree.
	DataType max() const;
	// Returns the minimum value of a node in the tree. You can assume that
	// this function will never be called on an empty tree.
	DataType min() const;
	// Returns the maximum depth of the tree. A tree with only the root node has a
	// depth of 0. You can assume that this function will never be called on an
	// empty tree.
	unsigned int depth() const;
	// You can print the tree in whatever order you prefer. However, methods such
	// as in-order or level-order traversal could be the most useful for debugging.
	void print() const;
	// Returns true if a node with the value val exists in the tree; otherwise,
	// it returns false.
	bool exists(DataType val) const;

	// Inserts the value val into the tree. Returns false if val already exists in
	// the tree, and true otherwise.
	bool insert(DataType val);
	// Removes the node with the value val from the tree. Returns true if
	// successful, and false otherwise.
	bool remove(DataType val);
};
#endif
```

Optional Part3: AVL Tree Implementation Using Inheritance [Bonus 10 marks]

This part is optional. If you choose to implement it, do not modify any of the code written for Part1 and Part2 of the assignment. Instead, copy over your Binary Search Tree implementation files into a new directory, and name them **`a3_binary_search_tree2.hpp/.cpp`**. Also, create two new files, **`a3_avl_tree.hpp/.cpp`**. Create a new project, and include the new files in the project.

Use inheritance to create **AVLTree** class as an extension of your **BinarySearchTree** class. That is, enter **class AVLTree : public BinarySearchTree {}** when declaring **AVLTree** in **a3_avl_tree.hpp** file.

Next, implement the new versions of **insert** and **remove** operations that keep the tree balanced. In the implementation of each function, first call the original implementation with **BinarySearchTree::insert** or **BinarySearchTree::remove**, to insert or remove the node from the tree, respectively. After calling the **BinarySearchTree** operations, you need to ensure that the tree is kept balanced by applying the appropriate AVL tree rotations: single-right, single-left, double-left-right, and double-right-left rotation. To that end, write and call a function called **balanceAVLTree**, which will check if the tree is balanced after insert or delete operation is called, and then call the appropriate tree rotation to balance the tree.

You may modify the **BinarySearchTree::Node** to include additional attributes that are needed for the **AVLTree** implementation, and modify **BinarySearchTree** implementation to include the updating of the additional attributes.

See the files in **a3_inheritance_examples.zip**, specifically **employee.hpp** and **employee.cpp**, for demonstrative examples on how to use inheritance in C/C++.

As additional reference, please review Chapter 11 of Main and Savitch, and consult the following URL: https://msdn.microsoft.com/en-us/library/84eaw35x.aspx

In your **main()** method, provide at least one test case for each AVL rotation type to demonstrate that your rotations have been implemented correctly. That is, for each test case, insert a number of nodes into the tree, and show the tree structure before and after the tree rotations are applied.

Deliverables

- Submit a **single zipped archive named studentid1_studentid2.zip** (e.g., 20000001_20000002.zip) with the files **a3_priority_queue.cpp** and **a3_binary_search_tree.cpp** included inside. Submit the zipped archive to the assignment dropbox on LEARN.

- If you choose to complete the optional task, then also include **a3_binary_search_tree2.hpp /.cpp** and **a3_avl_tree.hpp/.cpp** in your zipped archive.

- Do not include any other files in your zipped archive, such as your project files.

- Please write the names and IDs of both group members in the files that you submit as comments. Only one submission is required from each group.

- **The assignment is due on Mon Apr 6th by 23:55pm.**

- **No late submissions will be accepted.**