```
/*BST.h
 *
 *Dylan Jeffers
 *Tahmid Rahman
 *This file taken from Joshua Brody's CS31 Class
 *Taught Fall of 2014 @ Swarthmore College
 */

#ifndef BST_H_
#define BST_H_

#include "pair.h"
#include "library/queue.h"

/* An abstract (pure-virtual) representation of a binary-search-tree
 * (BST). Note that it is template definition based on a type for key
 * (K) and type for value (V).  BST is a dictionary structure; that is,
 * elements are indexed by key and all keys must be unique.
 */
template <typename K, typename V>
class BST {
  public:
    virtual ~BST() {};

    /* getSize - returns the size of the BST
     * @return int: the number of key-value pairs in the data structure
     */
    virtual int getSize() = 0;


    /* isEmpty  - returns true if the tree is empty
     * @return bool: true if there are no elements in the BST
     */
    virtual bool isEmpty() = 0;

    /* getHeight - returns a height for the tree (i.e., largest
     *   depth for any leaf node)
     * @return int: height of tree, -1 if tree is empty
     */
    virtual int getHeight() = 0;

    /* getMax - returns the largest key in the data structure
     * @return K: maximum key in the BST
     */
    virtual K getMax() = 0;

    /* getMin - returns the smallest key in the data structure
     * @return K: minimum key in the BST
     */
    virtual K getMin() = 0;

    /* insert - inserts the key-value pair into the tree
     * @param key - key for indexing the new element
     * @param value - value associated with the given key
     * @error runtime_error if they key already exists
     */
    virtual void insert(K key, V value) = 0;

    /* update - finds the element indexed by the given key and updates
```

```
     *          its value to the provided value parameter
     * @param key - key for finding the existing element
     * @param value - the new value to store for the given key
     * @error runtime_error if the key is not found in the BST
     */
    virtual void update(K key, V value) = 0;

    /* find - returns the value associated with the given key
     * @param key - index key for element to find
     * @error runtime_error if the key is not found in the BST
     * @return V: value associated with given key
     */
    virtual V find(K key) = 0;

    /* contains - returns true if there exists an element in the BST
     *   with the given key
     * @param key - index key to search for
     * @return bool: true if the given key exists in the BST
     */
    virtual bool contains(K key) = 0;

    /* remove - deletes the element with given key from the tree
     * @param key - index key to search for and remove
     * @error: runtime_error if they key is not found
     */
    virtual void remove(K key) = 0;

    /* The following are the iterators provided for the BST interface.
     *   Each returns a linear ordering of all elements in the BST stored in a
     *   Queue data structure.  Each key-value is stored as a Pair object
     * @return Queue< Pair<K,V> >*: a pointer to a dynamically allocated
     *   Queue with key-value pairs.  The caller is responsible for handling
     *   the heap memory deallocation
     */
    virtual Queue< Pair<K,V> >* getPreOrder() = 0;
    virtual Queue< Pair<K,V> >* getInOrder() = 0;
    virtual Queue< Pair<K,V> >* getPostOrder() = 0;
    virtual Queue< Pair<K,V> >* getLevelOrder() = 0;
};

#endif
```