## 4. Class Organization, Heaps, and BSTs (7 pts.)

**Complete your code in the provided `bst-heap.h` at the bottom of the file. You will **NOT** be able to compile and test this code because we do not provide the implementation of the base `BST<Key>` class.** Assume it is provided and works. We will visually grade your MaxHeap<Key> class assuming the BST class works.

Suppose you are provided a complete, templated binary search tree (BST) class: **`BST<Key>`.** It does not necessarily guarantee balance. We will assume type `Key` supports all the basic comparison operators: `<`, `>`, `==`, `!=`, etc.

You are now asked to write a **`MaxHeap<Key>`** class to implement a max heap (priority queue). It should use the **`BST<Key>`** class to implement it (though how to structure the relationship between these two is part of the question and left to you to decide). You may assume no duplicate keys are added to the heap.

Your `MaxHeap<Key>` must implement the following public interface using the standard definitions of the `push`, `pop`, and `top` operations. If **top()** or **pop()** is called on an empty heap, **throw std::out_of_range exception.**

```
template <typename Key>
class Heap /* your choice */
{
public:
    Heap();
    ~Heap();
    void push(const Key& newKey);
    void pop();
    const Key& top() const;  // throws std::out_of_range if empty
private:
    // add any data members or helper functions as necessary
};
```

### 4.1. Finish the implementation of the class / functions in the provided `bst-heap.h`.

- Your implementation should utilize/re-use as much (as reasonable) of the `BST<Key>` implementation to avoid unnecessary code. (There may be some duplication in your Heap implementation, but be judicious...if a BST operation can already accomplish a task, try to use it)
- Your runtime does not need to match that of a traditional Heap

### 4.2. Analyze the runtime of your `top()` implementation (show a very short justification or work).