CS271: Data Structures

Instructor: Dr. Stacey Truex

Project #4

This project is meant to be completed in groups. You should work in your Unit 3 groups. Solutions should be written in `C++` with one submission per group. Submissions should be a compressed file following the naming convention: `NAMES_cs271_project4.zip` where `NAMES` is replaced by the first initial and last name of each group member. For example, if Dr. Truex and Dr. Dolittle were in a group they would submit one file titled `STruexJDolittle_cs271_project4.zip`. **You will lose points if you do not follow the course naming convention**. Your `.zip` file should contain a *minimum* of 5 files:

1. `makefile`
2. `bst.cpp`
3. `test_bst.cpp`
4. `usecase.cpp`
5. `main.cpp`
6. `pdf of github commit history`

Additional files such as a `bst.h` header file or `README.md` are welcome. The above merely represent the minimum files required for project completion. Details for each are as follows.

Specifications

Binary Search Tree

Implement a `BST` template class using **two** templates - one for the data associated with each `BST` node and one for the key associated with each node. In your implementation, denote the data template first. Your class should, at a minimum, support the following operations:

- empty(): `bst.empty()` should indicate whether the binary search tree `bst` is empty. For example:

  ```
  BST<int, string> bst;
  if(bst.empty()){
      cout << "the bst is empty" << endl;
  }
  ```

  should result in the printing of the statement `"the bst is empty"`

- insert(`d`, `k`): `bst.insert(d, k)` should insert a node with data `d` and key `k` into the binary search tree `bst`. For example, using `bst` from above:

  ```
  bst.insert(271, "cs");
  ```

  should result in a bst with 1 node (the root) with a key of `"cs"` and data of `271`

- get(k): `bst.get(k)` should return the *data* associated with the *key* `k`. For example, using the tree `bst` from above:

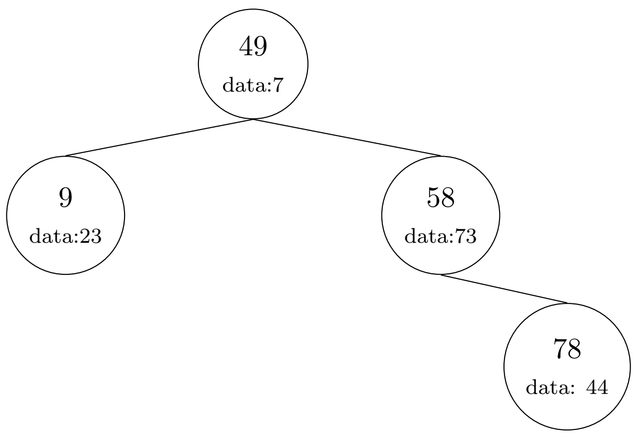
  ```
  cout << bst.get("cs") << endl;
  ```

  should result in the printing of the data 271

- remove(k): `bst.remove(k)` should delete the first (closest to the root) node in the tree `bst` with key `k`. For example:

  ```
  BST<int, int> bst2;
  bst2.insert(7, 49);
  bst2.insert(73, 58);
  bst2.insert(30, 72);
  bst2.insert(44, 78);
  bst2.insert(23, 9);
  bst2.remove(72);
  ```

  should result in the following BST:



- max_data(): `bst.max_data()` should return the *data* associated with the max *key* in the tree `bst`. For example, using `bst2` from above:

  ```
  cout << bst2.max_data() << endl;
  ```

  should result in the printing of the data value 44.

- max_key(): `bst.max_key()` should return the *key* associated with the max *key* in the tree `bst`. For example, using `bst2` from above:

  ```
  cout << bst2.max_key() << endl;
  ```

  should result in the printing of the key value 78.

- min_data(): `bst.min_data()` should return the *data* associated with the min *key* in the tree `bst`. For example, using `bst2` from above:

  ```
  cout << bst2.min_data() << endl;
  ```

  should result in the printing of the data value 23.

- min_key(): `bst.min_key()` should return the *key* associated with the min *key* in the tree `bst`. For example, using `bst2` from above:

  ```
  cout << bst2.min_key() << endl;
  ```

  should result in the printing of the key value 9.

- successor(k): `bst.successor(k)` should return the successor *key* in the tree `bst` for the key `k` (i.e., the smallest key in `bst` that is larger than `k`) if `k` is in the tree `bst`. For example:

  ```
  cout << bst2.successor(49) << endl;
  ```

  should result in the printing of the key value 58 while

  ```
  cout << bst2.successor(10) << endl;
  ```

  should print 0 as 10 is not a key in bst2

- in_order(): `bst.in_order()` should return a `string` of the keys in `bst` in *ascending* order separated by a single space. For example, using the `bst` from above:

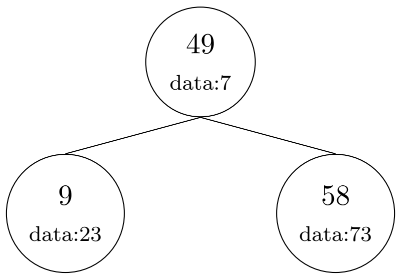
  ```
  cout << bst2.in_order() << endl;
  ```

  should print the string "9 59 58 78"

- trim(low, high): `bst.trim(low, high)` should trim the binary search tree `bst` so that the keys of every node lie in the interval [`low, high`]. Trimming the tree should not change the relative structure of the elements that will remain in the tree (i.e., any node's descendant should remain a descendant). For example, using the bst above:

  ```
  bst2.trim(9, 65);
  ```

  should result in the following BST:



Unit Testing

In addition to the functionality above, you are expected to implement a `to_string()` method which returns a string with the `keys` in the `BST` *separated by a single space* and ordered from top (root) to bottom (leaves) and left to right. For example, using the binary search tree `bst2` generated in the above specifications:

```
cout << bst2.to_string() << endl;
```

should result in the printing of the string "49 9 58". Your `to_string()` method will be required for your class to pass testing.

For each `BST` method included in your `BST` class, write a unit test method in a separate unit test file that *thoroughly* tests that method. Think, in addition to common cases: what are my boundary cases? edge cases? disallowed input? Each method should have *its own* test method.

An example test file `test_bst_example.cpp` has been provided and demonstrates (1) a general outline of what is expected in a test file and (2) a guide on how your projects will be tested after submission. The tests included in `test_bst_example.cpp` are not exhaustive. The unit testing in your `test_bst.cpp` file should be much more complete. Additionally, for grading purposes, your code will be put through significantly more thorough testing than what is represented by `test_bst_example.cpp`. Passing the tests in this example file should be viewed as a lower bound.

Documentation

The expectation of all coding assignments is that they are well-documented. This means that logic is documented with line comments and method pre- and post- conditions are properly documented immediately after the method's parameter list.

Pre-conditions and post-conditions are used to specify precisely what a method does. However, a pre-condition/post-condition specification does not indicate how that method accomplishes its task (if such commenting is necessary it should be done through line level comments). Instead, pre-conditions indicate what must be true before the method is called while the post-condition indicates what will be true when the method is finished.

Use Case

Finally, use your `BST` to solve the following problem:

You are given a csv file in which each line represents a hex, bin pair where each bin is a 4-bit binary and each hex is the corresponding 1 digit hexidecimal. Use the csv to build a BST using your `BST` class where each node in the tree is characterized by: (1) a key of the 4-bit binary and (2) data denoting the corresponding hexidecimal conversion. Your program should then accomplish the following:

- Ask the user for a binary value for conversion.
- Convert the binary value to the corresponding hexidecimal.
- Display the result.

Two examples might be as follows:

```
Enter binary representation for conversion:
111010100101
Hexidecimal representation of 111010100101 is EA5

Enter binary representation for conversion:
110101
Hexidecimal representation of 110101 is 35
```

The csv file for building your BST has been provided. Note that the binary representation provided by the user may not be in multiples of 4. In that case, pad the front of the input with additional zeros.

Your solution should be implemented in `usecase.cpp` using the following two functions:

```
BST<D,K>* create_bst(string fname)
string convert(BST<D,K>* bst, string bin)
```

where `fname` is the name of the csv file containing the binary, hexidecimal pairs. Your generated binary search tree should then be used with the `convert` function where `bst` is the tree from the `create_bst` function and `bin` is the entered binary representation you wish to convert. Your function should return the `string` corresponding to the hexidecimal conversion of this `bin` *based on the data from the csv.* Note that your use case code will only be tested when the templates are both set to `string`.

In your `main.cpp` file, your `main` function should include at least one example test case demonstrating the accuracy of your solution which allows for user input from the terminal.

Makefile

With each project you should be submitting a corresponding makefile. Once unpacking your `.zip` file, the single command `make` should create a `test` executable **and** a `usecase` executable. The command `./test` should then run all the unit tests in your `test_bst.cpp` file evaluating your `BST` class. The command `./usecase` should run the example test case in your `main.cpp` file demonstrating the accuracy of your conversion solution.

Efficiency

Each project in this course will additionally be evaluated for efficiency. Each method detailed in the BST class methods section of this document will be called 1 time using a very large example. The total time to execute all methods will be clocked.

Peer Evaluation

Each partner should additionally complete a pair evaluation form reflecting on the performance of both themselves and their partner with respect to professionalism and effort on the project. Reflections should be submitted individually, separate from the project file. Note that reflections will always remain confidential.

Rubric

Code	**Completeness** met submission requirements	15
	Correctness passes unit testing	48
	Correctness implementation deductions ex: incomplete destructor	0
Usecase	**Correctness** passes unit testing	11
Efficiency	time test	10
Documentation	detailed comments pre- and post-conditions	6
Testing	thoroughness of unit tests	10