CS 271 Project 4 – Another Way of Searching (Binary Search Tree Implementation)

Instructor: Flannery Currin

Due: October 29th, 2025 by 9:00 AM

1 Learning Goals

Binary search trees allow us to bring together the concepts we have covered with heaps and hash tables. We can use BSTs as both dictionaries and priority queues. Specifically, after working on this project you should:

- Understand the relationship between the data of an element and its associated key in a binary search tree
- Know how to maintain the binary search tree property
- Know how to navigate a binary search tree (e.g., to find the min or max element)
- Know how to write a template class with two template types

2 Project Overview

You should complete this project in a group as assigned on Canvas and in class. You are individually responsible for the learning goals above (dividing and conquering may not be the most effective strategy here). Use the Canvas guide on using Github to help manage version control. You may discuss this assignment with your partner(s), the course TA or instructor, but the work you submit must be your group members' own work. You may use your previous projects in this class as a reference for how to write a templated class, and external guides to learn how to work with multiple templated types.

2.1 BST Class

Implement a `BST` template class using two template types - one for the *data* associated with each BST node and one for the *key* associated with each node. In your implementation, the data template type should come first (i.e., `<D, K>` where `D` is the type of the data and `K` is the type of the key). 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 print `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 the BST above:

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

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

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

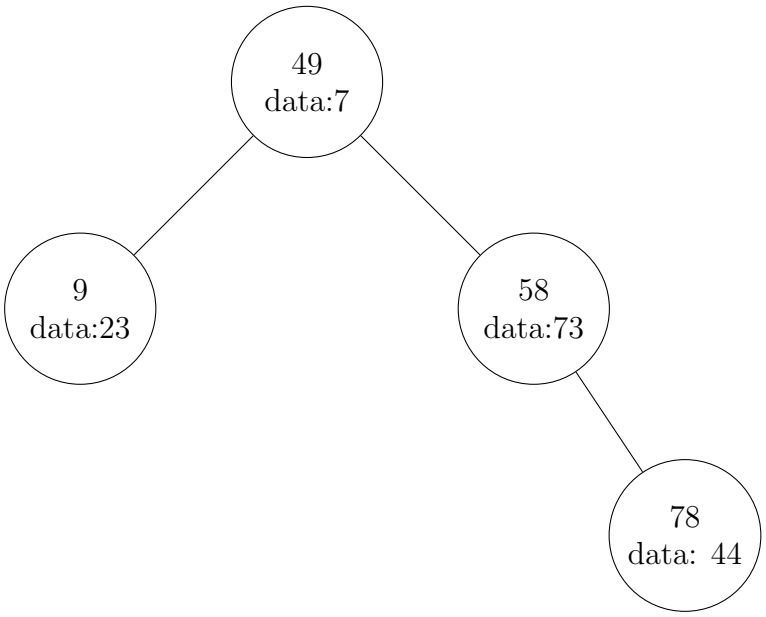
```
cout << bst.get(``cs'') << endl;
```

  should print 271.

- `remove(k)`: `bst.remove(k)` should remove the first (closest to the root) node with key `k` in the binary search tree `bst`. 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 produce the following BST:



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

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

  should print 44.

- `max_key()`: `bst.max_key())` should return the **max key** in the binary search tree `bst`. For example, using the BST above:

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

  should print 78.

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

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

  should print 23.

- `min_key()`: `bst.min_key())` should return the **min key** in the binary search tree `bst`. For example, using the BST above:

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

  should print 9.

- `successor(k)`: `bst.successor(k))` should return the *key* of the successor in the binary search tree `bst` for key `k` (i.e., the smallest key in bst that is larger than `k`). If no such successor exists, return 0. For example, using the BST above:

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

  should print 58.

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

  should print 0 as 10 is not a key in `bst2`.

- `to_string()`: `bst.to_string()` 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 above bst:

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

  should print: 49 9 58 78.

- `in_order()`: `bst.in_order()` returns a string with the *keys* in the BST separated by a single space and *in ascending order*. For example, using the above bst:

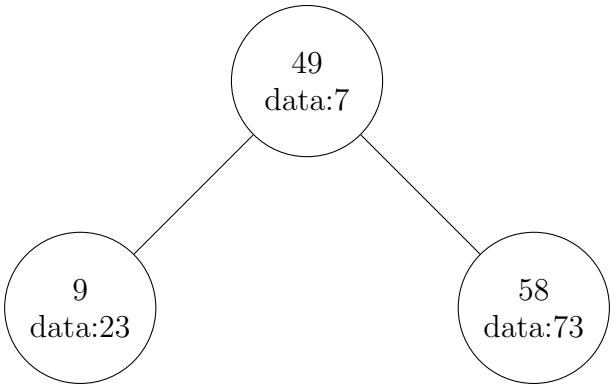
```
cout << bst2.to_string() << endl;
```

  should print: 9 49 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 produce the following BST:



2.2 Unit Testing

Test each `BST` function thoroughly, using the provided test file `test_bst_example.cpp` as a guide and starting point.

2.3 Usecase

Finally, use your `BST` class in `usecase.cpp` to solve the following problem. Given a txt file (`binhex.txt` in which each line represents a hex, bin pair where each bin is a 4-bit binary number and each hex is the corresponding 1-digit hexadecimal number, 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 hexadecimal conversion. Your program should then:

- Ask the user for a binary value for conversion.
- Convert the binary value to the corresponding hexadecimal.
- Display the result and return the string containing the hexadecimal.

Two examples might be as follows:

```
Enter binary representation for conversion:
111010100101
Hexadecimal representation of 111010100101 is EA5
Enter binary representation for conversion:
110101
Hexadecimal representation of 110101 is 35
```

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 functions:

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

where `fname` is the name of the txt file containing the binary, hexadecimal 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 hexadecimal conversion of this bin based on the data from the txt. Note that your use case code will only be tested when the templates are both set to string.

In your `usecase.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.

3 Project Submission

On Canvas, before the deadline, you will submit **a zip** containing:

- Your `BST.h` and/or `BST.cpp` files.
- `bst_test.cpp` with your unit tests
- `usecase.cpp` with your usecase implementation and tests
- A makefile to compile your project
- Any additional files you created that are needed to compile your project using your makefile

4 Grading Scheme

The out-of-class submission will be graded using the following scheme:

Criteria	**Description**	**Points**
Completeness	Meets submission requirements (files, documentation, etc.)	2
Correctness	Passes Dr. Currin's extended tests and follows specs	4
Usecase	Contains required components and thorough testing	2
Testing	Tests should cover a range of types and cases, and logic should be sound	2
Total		**10**

The in-class component is worth 20 points. You should not need to study extra or memorize your out-of-class work for the in-class component. Working on the out-of-class component with the learning goals in mind is your best preparation. The in-class project component lets you test how prepared you are to apply the concepts covered by the project to a new scenario.