EEE2020-03 **Spring 2021**

Assignment 5: Binary Search Tree

Due: Sunday, May 30, 2021, 11:59PM

1 Introduction

- The objective of this assignment is to implement a binary search tree ADT.
- In this assignment, you will have to complete the missing parts of `class tree_t` and `class bst_t` based on the lecture slides of `5_trees.pdf`.
- In Ubuntu or Mac terminal, use `wget` to download a copy of skeleton code compressed into `bst.tar`.

```
$ wget https://icsl.yonsei.ac.kr/wp-content/uploads/bst.tar
```

- Decompress the `tar` file, and go to the `bst/` directory. Typing `ls` should show the following list of files.

```
$ tar xf bst.tar
$ cd bst/
$ ls
bst.cc  data.h  input.cc  main.cc   node.h  tree.cc
bst.h   input   input.h   Makefile  tar.sh  tree.h
```

- From the listed files, you have to work on four files, `tree.h`, `tree.cc`, `bst.h`, and `bst.cc`.
- You are allowed to change other files for your own testing and validation, but they will revert to the original state when your assignment is graded.
  - `tree.*` define a base class of binary tree, `class tree_t`.
  - `bst.*` define a derived class of binary search tree, `class bst_t`.
  - `node.h` has the definition of tree node, `class node_t`.
  - `input.*` files define `class input_t` that handles file operations. This class loads a file named `input` containing 100 unsigned integer numbers. The numbers are used in `main()` to test the operations of BST.
  - `main.cc` has the `main()` function.
- To compile the code, type `make` in the terminal. `Makefile` that comes along with the skeleton code has automated all the compiling scripts for you.

```
$ make
g++ -Wall -Werror -g -o main.o -c main.cc
g++ -Wall -Werror -g -o bst.o -c bst.cc
g++ -Wall -Werror -g -o input.o -c input.cc
g++ -Wall -Werror -g -o tree.o -c tree.cc
g++ -o bst main.o bst.o input.o tree.o
```

- Executing the skeleton code prints the following output.

```
$ ./bst
height() = 0
size() = 0

height() = 0
size() = 0
```

2 Implementation

- `node.h` has the definition of tree node, `class node_t`.
- The node implementation is self-contained in the header and not split to a source file. The header is complete, and you do not have to touch anything in this file.
- `class node_t` is simply a group of `*left`, `*right`, and `data`. The first two are pointers to left and right children nodes, and the last variable is the actual data enclosed in the node.

```
/* node.h */

#ifndef __NODE_H__
#define __NODE_H__

#include "data.h"

class node_t {
public:
    // Node constructors
    node_t(void) : left(0), right(0) { }
    node_t(const data_t &m_data) : left(0), right(0), data(m_data) { }

    node_t *left, *right;       // Pointers to left and right children
    data_t data;                // Node data;
};

#endif
```

- The next shows `tree.h` that defines `class tree_t`, the base class of binary tree.
- The class definition in the skeleton code includes only public functions. You will have to write recursive functions as needed either in the `protected` or `private` sections of the BST class.
- Writing recursive functions is not mandatory. If you can figure out a way to implement the BST class without recursive functions, it is acceptable as far as it works.
- This class has two `protected` variables, `num_elements` and `root`. The first variable indicates the number of elements (or nodes) in the tree, and the second one is a pointer to the root node.

```
/* tree.h */

#ifndef __TREE_H__
#define __TREE_H__

#include <cstdlib>
#include "node.h"

// Binary tree base class
class tree_t {
public:
    tree_t(void);
    virtual ~tree_t(void);

    // Get the number of nodes in the tree.
    size_t size(void) const;
    // Get the tree height.
    int height(void) const;
    // Get the minimum element in the tree.
```

```
    const data_t& find_min(void) const;
    // Get the maximum element in the tree.
    const data_t& find_max(void) const;
    // Return true if the tree contains the data.
    virtual bool contains(const data_t &m_data);
    // Add a new element to the tree.
    virtual void insert(const data_t &m_data) = 0;
    // Remove the element in the tree if it has one.
    virtual void remove(const data_t &m_data) = 0;
    // Remove all nodes in the tree.
    void clear(void);
#ifdef DEBUG
    // Print node data in preorder.
    virtual void print(void) const;
#endif

private:
#ifdef DEBUG
    void print(const node_t *m_node, const int m_depth = 0) const;
#endif

/*************************
 * EEE2020: Assignment 5 *
 *************************/

protected:
    size_t num_elements;    // Number of elements in the tree
    node_t *root;           // Pointer to the root node

/*********************
 * End of Assignment *
 *********************/
};

#endif
```

- Optionally, a `print()` function is provided for debugging purpose. This function prints the organization of tree nodes. A complete implementation of `print()` is included in the skeleton code.
- For instance, suppose the root has a value of 45 and two children, 23 on the left and 71 on the right. Calling `print()` shows the tree structure rotated 90 degrees counterclockwise as follows.

```
     ...71
...45
     ...23
```

- The next shows the contents of `tree.cc`. The skeleton code includes only the class constructor of `tree_t` and the `print()` function. All others are for the assignment.
- If you added recursive functions to `class tree_t` in `tree.h`, you will have to write their implementations in `tree.cc` as well. The skeleton code shows only the public functions of `tree_t`.
- Functions with non-`void` returns in the skeleton code are written to return random values to avoid compile errors. You have to correct them based on your implementations.

```
/* tree.cc */

#include <cstdlib>
#include <new>
#include "tree.h"


/************************
 * EEE2020: Assignment 5 *
 ************************/

// Destructor
tree_t::~tree_t(void) {
    /* Assignment */
}

// Get the number of nodes in the tree.
size_t tree_t::size(void) const {
    /* Assignment */
    return 0;           // Correct this.
}

// Get the tree height.
int tree_t::height(void) const {
    /* Assignment */
    return 0;           // Correct this.
}

// Get the minimum element in the tree.
const data_t& tree_t::find_min(void) const {
    /* Assignment */
    return root->data;  // Correct this.
}

// Get the maximum element in the tree.
const data_t& tree_t::find_max(void) const {
    /* Assignment */
    return root->data;  // Correct this.
}

// Return true if the tree contains the data.
bool tree_t::contains(const data_t &m_data) {
    /* Assignment */
    return false;       // Correct this.
}

// Remove all nodes inn the tree.
void tree_t::clear(void) {
    /* Assignment */
}

/*********************
 * End of Assignment *
 *********************/
```

```
// Constructor
tree_t::tree_t(void) :
    num_elements(0),
    root(0) {
    // Nothing to do
}

#ifdef DEBUG
#include <iostream>
void tree_t::print(void) const { print(root); }

void tree_t::print(const node_t *m_node, const int m_depth) const {
    if(!m_node) { return; }
    print(m_node->right, m_depth+1);
    if(m_node->left && !m_node->right)  { std::cout << std::endl; }
    for(int d = 0; d < m_depth; d++) { std::cout << "      ";  }
    std::cout << "..."; std::cout << m_node->data << std::endl;
    print(m_node->left,  m_depth+1);
    if(!m_node->left && m_node->right)  { std::cout << std::endl; }
}
#endif
```

- The next code box is the `bst.h` file. This file defines `class bst_t` as a derived class of `tree_t`.
- The `bst_t` class reuses most of functions provided by `tree_t`. It defines only `insert()` and `remove()` as its own functions. Note that these two functions are defined with `virtual` keywords in the base class of `tree_t`.
- Similar to `tree_t`, the class in the skeleton code has only public functions. You may have to write recursive versions of `insert()` and `remove()` in the `private` section.

```
/* bst.h */

#ifndef __BST_H__
#define __BST_H__

#include "tree.h"

// Binary search tree (BST) based on the binary tree base class
class bst_t : public tree_t {
public:
    bst_t(void);
    ~bst_t(void);

    // Add a new element to the tree.
    void insert(const data_t &m_data);
    // Remove the element in the tree if it has one.
    void remove(const data_t &m_data);

/************************
 * EEE2020: Assignment 5 *
 ************************/


/*********************
 * End of Assignment *
 *********************/
};

#endif
```

- The next shows `bst.cc` that defines the implementation of `bst_t`.
- The skeleton code provides only the constructor and destructor of the class. All others are for the assignment.
- You do not have to implement copy constructors or assignment operators for both `bst_t` and `tree_t`. They are not used in this homework.

```
#include <cstdlib>
#include <new>
#include "bst.h"

/************************
 * EEE2020: Assignment 5 *
 ************************/

// Add a new element to the tree.
void bst_t::insert(const data_t &m_data) {
    /* Assignment */
}

// Remove the element in the tree if it has one.
void bst_t::remove(const data_t &m_data) {
    /* Assignment */
}

/*********************
 * End of Assignment *
 *********************/




// Constructor
bst_t::bst_t(void) { /* Nothing to do */ }

// Destructor
bst_t::~bst_t(void) { /* Nothing to do */ }
```

- Your are allowed to change `main()` to further test and validate your BST implementation between the lines of `* TEST YOUR CODE HERE FOR VALIDATION *` and `* END OF TESTING *`. However, this file will revert to the original state when your assignment is graded.

```
/* main.cc */

#include <iostream>
#include "input.h"
#include "bst.h"

using namespace std;

// Print the information of BST
void print(bst_t &bst) {
    cout << "height() = "   << bst.height()   << endl;
    cout << "size() = "     << bst.size()     << endl;
    if(bst.size()) {
        cout << "find_min() = " << bst.find_min() << endl;
        cout << "find_max() = " << bst.find_max() << endl;
    }
```

```
#ifdef DEBUG
    cout << "nodes: "       << endl; bst.print();
#endif
    cout << endl;
}

int main(int argc, char **argv) {
    // Define a BST.
    bst_t bst;

    /************************************
     * TEST YOUR CODE HERE FOR VALIDATION *
     ************************************/
    bst.insert(45);
    bst.insert(23);
    bst.insert(71);
    bst.insert(34);
    bst.insert(55);
    bst.insert(10);
    bst.insert(83);
    bst.insert(7);
    bst.insert(30);
    bst.insert(64);
    bst.insert(15);
    bst.insert(95);
    bst.insert(39);
    bst.insert(77);
    bst.insert(37);
    bst.insert(41);
    bst.insert(80);
    bst.insert(59);
    bst.insert(2);
    print(bst);

    bst.remove(59);
    bst.remove(55);
    bst.remove(71);
    bst.remove(15);
    bst.remove(95);
    bst.remove(2);
    print(bst);
    /******************
     * END OF TESTING *
     ******************/




    /* ************************************************
     * WARNING: DO NOT MODIFY THE CODE BELOW THIS LINE *
     ************************************************/

    // Proceed?
    if(argc != 2) { return 0; }

    // Empty the BST.
    cout << "-----------------" << endl
         << "Test #1: clear()" << endl
         << "-----------------" << endl;
```

```
    bst.clear();
    print(bst);


    // Add elements read from the input file to the BST.
    cout << "-----------------" << endl
         << "Test #2: insert()" << endl
         << "-----------------" << endl;
    input_t input(argv[1]);
    for(size_t i = 0; i < input.size(); i++) { bst.insert(input[i]); }
    print(bst);


    // Check if certain elements are contained in the BST.
    cout << "--------------------" << endl
         << "Test #3: contains()" << endl
         << "--------------------" << endl;
    data_t d;
    cout << "contains(" << (d = 497)  << ") = "
         << (bst.contains(d) ? "yes" : "no") << endl;
    cout << "contains(" << (d =  24)  << ") = "
         << (bst.contains(d) ? "yes" : "no") << endl;
    cout << "contains(" << (d = 169)  << ") = "
         << (bst.contains(d) ? "yes" : "no") << endl;
    cout << "contains(" << (d =  15)  << ") = "
         << (bst.contains(d) ? "yes" : "no") << endl;
    cout << "contains(" << (d = 275)  << ") = "
         << (bst.contains(d) ? "yes" : "no") << endl;
    cout << "contains(" << (d = 387)  << ") = "
         << (bst.contains(d) ? "yes" : "no") << endl;
    cout << "contains(" << (d =   9)  << ") = "
         << (bst.contains(d) ? "yes" : "no") << endl;
    cout << "contains(" << (d = 400)  << ") = "
         << (bst.contains(d) ? "yes" : "no") << endl;
    cout << "contains(" << (d = 251)  << ") = "
         << (bst.contains(d) ? "yes" : "no") << endl;
    cout << endl;


    // Remove elements in the BST.
    cout << "------------------" << endl
         << "Test #4: remove()" << endl
         << "------------------" << endl;
    bst.remove(497);
    bst.remove(24);
    bst.remove(169);
    bst.remove(15);
    bst.remove(275);
    bst.remove(387);
    bst.remove(9);
    bst.remove(400);
    bst.remove(251);
    print(bst);


    // Repeat all functions once again.
    cout << "-----------------------------------" << endl
         << "Test #5: clear(), insert(), remove()" << endl
         << "-----------------------------------" << endl;
```

```
    bst.clear();
    for(size_t i = input.size(); i > 0; i--) { bst.insert(input[i-1]); }
    bst.remove(322);
    bst.remove(340);
    bst.remove(203);
    print(bst);


    return 0;
}
```

- Your assignment is graded based on what you get from the testing part of `main()`. Results above the line of `* WARNING: DO NOT MODIFY THE CODE BELOW THIS LINE *` will be not considered for grading.
- **Test #1:** The first part of testing clears `bst` by calling `bst.clear()`.
- **Test #2:** The test code reads an input data file (i.e., `input` in the `bst/` directory) that contains 100 unsigned integer numbers between 0 and 500. File handling is already implemented, and you may disregard how it works. Numbers in the file are inserted into the BST via repeated calls of `bst.insert()`.
- **Test #3:** It calls a set of `contains()` functions to check if `bst` contains certain values.
- **Test #4:** This part of code tests the `remove()` function.
- **Test #5:** Lastly, `clear()`, `insert()`, and `remove()` functions are all tested once again. This time, 100 numbers in the `input` file are inserted in the reverse order, and thus a different tree shape is created.
- If both `class tree_t` and `class bst_t` are correctly implemented, executing the code will produce the following output.

```
$ ./bst
height() = 4
size() = 19
find_min() = 2
find_max() = 95

height() = 4
size() = 13
find_min() = 7
find_max() = 83
```

- To enable the `print()` function of `tree_t`, you have to clean up the build files and recompile the code by typing `make "OPT=-DDEBUG"`. Note that are double `D`'s in the optional flag.

```
$ make clean
rm -f bst.o input.o main.o tree.o bst

$ make "OPT=-DDEBUG"
g++ -Wall -Werror -g -o bst.o -c bst.cc -DDEBUG
g++ -Wall -Werror -g -o input.o -c input.cc -DDEBUG
g++ -Wall -Werror -g -o main.o -c main.cc -DDEBUG
g++ -Wall -Werror -g -o tree.o -c tree.cc -DDEBUG
g++ -o bst bst.o input.o main.o tree.o

$ ./bst
height() = 4
size() = 19
find_min() = 2
find_max() = 95
```

```
nodes:
                  ...95
            ...83
                        ...80
                  ...77

      ...71

                  ...64
                        ...59
            ...55

...45
                        ...41
                  ...39
                        ...37
            ...34
                  ...30
      ...23
                  ...15
            ...10

                  ...7
                        ...2

height() = 4
size() = 13
find_min() = 7
find_max() = 83
nodes:

            ...83
                  ...80
      ...77
            ...64
...45
                        ...41
                  ...39
                        ...37
            ...34
                  ...30
      ...23

            ...10
                  ...7
```

- To test the program with the `input` data file, add `input` to the run command as follows.

```
$ make clean
rm -f bst.o input.o main.o tree.o bst

$ make
g++ -Wall -Werror -g -o bst.o -c bst.cc
g++ -Wall -Werror -g -o input.o -c input.cc
g++ -Wall -Werror -g -o main.o -c main.cc
g++ -Wall -Werror -g -o tree.o -c tree.cc
g++ -o bst bst.o input.o main.o tree.o
```

```
$ ./bst input
height() = 4
size() = 19
find_min() = 2
find_max() = 95

height() = 4
size() = 13
find_min() = 7
find_max() = 83

----------------
Test #1: clear()
----------------
height() = -1
size() = 0

-----------------
Test #2: insert()
-----------------
height() = 13
size() = 100
find_min() = 9
find_max() = 497

-------------------
Test #3: contains()
-------------------
contains(497) = yes
contains(24) = yes
contains(169) = yes
contains(15) = no
contains(275) = yes
contains(387) = yes
contains(9) = yes
contains(400) = no
contains(251) = yes

-----------------
Test #4: remove()
-----------------
height() = 11
size() = 93
find_min() = 25
find_max() = 493

-----------------------------------
Test #5: clear(), insert(), remove()
-----------------------------------
height() = 10
size() = 97
find_min() = 9
find_max() = 497
```

- Even if your code compiles and runs to completion with correct results, you need to double-check the code with `valgrind` to confirm no memory leaks.

```
$ valgrind ./bst input
```

```
...

==2020== All heap blocks were freed -- no leaks are possible

...
```

- In Mac OS, `valgrind` is not well supported. You may use `leaks` instead to test memory leaks.

```
$ leaks --atExit -- ./bst input

...

Process 2020: 0 leaks for 0 total leaked bytes.

...
```

3 Submission

- When the assignment is done, execute the `tar.sh` script in the `bst/` directory.
- It will compress the `bst/` directory into a `tar` file named after your student ID such as `2020140000.tar`.

```
$ ./tar.sh
rm -f main.o bst.o input.o tree.o bst
bst/
bst/bst.cc
bst/tree.h
bst/._node.h
bst/main.cc
bst/Makefile
bst/node.h
bst/bst.h
bst/._bst.h
bst/data.h
bst/input
bst/input.cc
bst/tar.sh
bst/tree.cc
bst/._tree.h
bst/input.h

$ ls
2020140000.tar  bst.h   input     input.h  Makefile  tar.sh   tree.h
bst.cc          data.h  input.cc  main.cc  node.h    tree.cc
```

- Upload the tar file (e.g., `2020140000.tar`) on LearnUs. Do not rename the `tar` file or C++ files included in it.

4 Grading Rules

- The following is the general guideline for grading. 30-point scale will be used for this assignment. The minimum score is zero, and negative scores will not be given. Grading rules are subject to change, and the grader may add a few extra rules for fair evaluation of students' efforts.

  -3 points: A submitted `tar` file is renamed and includes redundant tags such as `hw5`, student name, etc.

  -5 points: A program code does not have sufficient amount of comments. Comments in the skeleton code do not count. You must make an effort to clearly explain what each part of your code intends to do.

-5 points each: There are total five sets of testing in `main()`, such as `Test #1: clear()`, $\cdots$ For each incorrect block, 5 points will be deducted.

-15 points: The code compiles and runs, but it has memory leaks or crashes at the end with errors.

-25 points: The code does not compile or fails to run (e.g., no or completely wrong results), but it shows substantial efforts to complete the assignment.

-30 points: No or late submission. The following cases will also be regarded as no submissions.

* Little to no efforts in the code (e.g., submitting nearly the same version of code as the skeleton code) will be regarded as no submission. Even if the code is incomplete, you must make substantial amount of efforts to earn partial credits.
* Fake codes will be regarded as cheating attempts and thus not graded. Examples of fake codes are i) hard-coding a program to print expected outputs to deceive the grader as if the program is correctly running, ii) copying and pasting random stuff found on the Internet to make the code look as if some efforts are made. More serious penalties may be considered if students abuse the grading rules.

Final grade = F: The submitted code is copied from someone else. All students involved in the incident will be penalized and given F for the final grades irrespective of assignments, attendance, etc.

- Your teaching assistant (TA) will grade your assignments. If you think your assignment score is incorrect for any reasons, feel free to discuss your concerns with the TA. In case no agreement is made between you and the TA, elevate the case to the instructor to review your assignment. Refer to the course website for the contact information of TA and instructor: `https://icsl.yonsei.ac.kr/eee2020`
- Begging partial credits for no valid reasons will be treated as a cheating attempt, and such a student will lose all scores of the assignment.