Sabancı University

Faculty of Engineering and Natural Sciences
CS204 Advanced Programming
Fall 2022

Homework 4 – ActivityBST class

Due: 29/11/2022, Tuesday, 11:55

PLEASE NOTE:

Your program should be a robust one such that you have to consider all relevant programmer mistakes and extreme cases; you are expected to take actions accordingly!

You HAVE TO write down the code on your own.
You CAN NOT HELP any friend while coding.
Plagiarism will not be tolerated!!

Introduction

In the last homework, you implemented an activity binary search tree in a functional manner. In this homework, you will make this activity binary search tree into the class `ActivityBST`.

You will find on SUCourse the implementations of all the functions from the last homework, and you may use them for the implementation of this homework if you want.

Your submission to SUCourse will consist of two files only:

- ActivityBST.h: a file containing the following:
    1. The `tree_node` struct (its code is given in the next section),
    2. the declaration of your ActivityBST class, and
    3. the declaration of any other functions you use in your solution.
- ActivityBST.cpp: a file containing the definitions (implementations) of your class functions and any other functions you declared in ActivityBST.h.

Rules

Before getting into the functions you will implement, let's lay down some rules.

1. You are allowed to implement additional functions besides the ones described in this document in the `ActivityBST` files you will write if they help you with your solution.
2. You are NOT allowed to use vectors, arrays, or any other containers to store values anywhere in the code.

3. You are allowed to use libraries within the C++ standard library. These are libraries like `<string>`, `<sstream>`, `<time>`, etc.
4. You are NOT allowed to use external libraries that are not in the C++ standard library. For example, the `strutils.h` library from CS 201.
5. Do not use Turkish letters in the code or comments. It causes some issues with the grading software.

`ActivityBST` description

Each `ActivityBST` object will store the activities that the user will do during the day and their times in a binary search tree. The tree will be sorted based on the times of the activities. The following is an example of an ActivityBST:

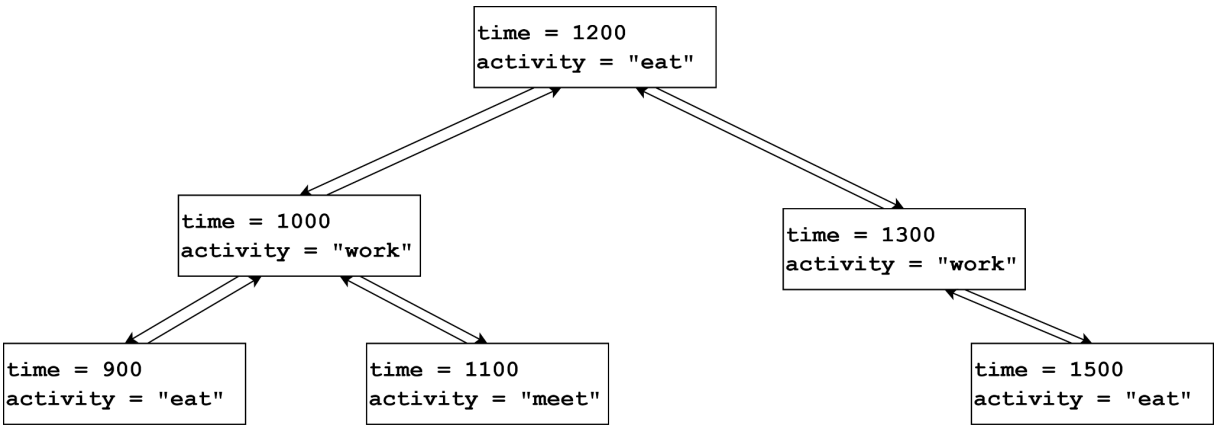

Figure 1: an activity tree representing a daily schedule. Each node contains an integer `time` representing the time of the activity in 24-hour notation (without the colon), and a string `activity` containing the activity name. The tree is a binary search tree sorted according to `time` integers.

The following is the definition of the tree node structs. **Please add this code to your ActivityBST.h file**:

```
struct tree_node {
   int time;
   string activity;
   tree_node *right;
   tree_node *left;
   tree_node *parent;

   tree_node(int t, const string &a) : time(t), activity(a),
                                       right(nullptr), left(nullptr),
                                       parent(nullptr) {}
};
```

You don't need to add any code pertaining to the tree_node struct to your ActivityBST.cpp file.
Note: you are not allowed to change this struct in any way. That includes adding other constructors.

Each tree node contains an integer `time` with the time of the activity, a string `activity` with its name, pointers at the node's right and left children, as well as a pointer at the parent of the node. If a node does not have a parent, its parent pointer should be set to nullptr.

Since this tree is a binary search tree, it must be sorted based on some value. **The nodes in this tree will be sorted based on the time integer.**

Data members:

The `ActivityBST` class you write must have only ONE data member, which is private:

```
tree_node*  root; // the root of the BST
```

You are NOT allowed to add any additional data members to the class.

Functions:

The following are the member functions of the class `ActivityBST`. Note that you may add more functions to the class if they help you implement the required functions.

Default constructor

A default constructor that will initialize the tree to an empty tree.

Usage example:

```
ActivityBST bst; // default constructor is called
```

Copy constructor

Copies an existing ActivityBST to the calling object.

Usage example

```
ActivityBST abst1;
ActivityBST abst2(abst1); // copy constructor is called
```

Destructor

Deletes the tree.

add(int time, const string& activity)

The function takes an integer representing a time, and a name for an activity, and will add a tree node object representing this activity to the ActivityBST. Remember that this is a binary search tree sorted based on the time integer.

Unlike the last homework's add_activity function, this add function does not print anything to standard output. It also returns void.

Error checking

You do not need to make any error checks for the `activity` string. However, you must check that the `time` value passed is legal. A time value is legal if it satisfies two conditions:

1. The minutes part is in the range of 0-59.
2. The hours part is in the range of 0-23.

If the passed time is illegal, then you will not do anything to the tree (nothing will be added). You don't have to print anything to standard output if a node isn't added successfully.

Assumptions

You may assume that there will never be two calls to "add" with the same time value. In other words, once an activity is added at time X, no other activities will be added with the same time value.

Example usage

```
ActivityBST a;
a.add(1000, "eat");
```

operator=

Left-hand side: ActivityBST object
Right-hand side: ActivityBST object

Assignment operator that assigns the right-hand-side object to the left-hand side object. Note that this operator must allow operator cascading.

Example usage

```
ActivityBST a1, a2, a3;
a3 = a2 = a1; // operator= is called twice
```

operator+

Left-hand side: ActivityBST object
Right-hand side: ActivityBST object

Returns a new ActivityBST object that contains all the activities of the right-hand side ActivityBST, and all the activities of the left-hand side ActivityBST. You may assume that none of the activities in the left-hand side tree will have a time value equal to an activity in the right-hand side tree, and vice versa.

Remember that the result of this operation is an ActivityBST, so its activities must be sorted based on their time value.

Example Usage

```
ActivityBST a1, a2;
ActivityBST a3 = a1 + a2; // operator+ is called
```

operator+=

Left-hand side: ActivityBST
Right-hand side: ActivityBST

Adds all the activities in the right-hand side ActivityBST to the left-hand side ActivityBST. Must allow operator cascading. You may assume that none of the activities in the left-hand side tree will have a time value equal to an activity in the right-hand side tree, and vice versa.

Remember to make sure the activities of the left-hand side object remain sorted after adding the activities of the right-hand side.

Example usage

```
ActivityBST a1, a2;
a1 += a2; // a1 has the activities of a1 and a2
ActivityBST b1, b2, b3;
b1 += b2 += b3; // b2 has the activities of b2 and b3
                // b1 has the activities of b1, b2, and b3
```

Free functions

You only have to define a single free function for this homework. Remember that a free function is one that is not a member of a class.

You must place its declaration in the file ActivityBST.h, and its definition (implementation) in the file ActivityBST.cpp.

operator<<

Left-hand side: ostream
Right-hand side: ActivityBST

Print all the activities of the ActivityBST object on the right-hand side of the operator to the output stream that is the left-hand side operand of this operator. The operator must allow operator cascading.

Error checking
There is no error checking you need to do for the input.

Output
This function prints *to the left-hand side output stream* the start times of all the activities in the tree, each on a separate line of the form:

```
[HH:MM] - act
```

where `HH:MM` is the start time of the activity, and `act` is its name.

Only print the start times of activities in the tree. If there are no activities in the tree, that operator call should not print anything.

Calling this operator on the tree in Figure 1 will print:

```
[09:00] - eat
[10:00] - work
[11:00] - meet
[12:00] - eat
[13:00] - work
[15:00] - eat
```

Example usage

```
ActivityBST a, b;
ofstream fout("output.txt");
fout << a; // prints a to a file
cout << a << b; // prints a and b to standard output
```

Sample main programs

The following are two examples of a main programs that use the ActivityBST class. These main programs use the functions required of you to implement. During grading, we will use different main programs for each test case, and each will use different functions and different inputs. However, all the test cases will use the *add* function, so **if you don't implement the *add* function correctly, you will get 0 for all the test cases.**

Sample main program 1

`main.cpp`

```
int main() {
   ActivityBST uni_calendar;
   uni_calendar.add(1000, "meeting");
   uni_calendar.add(1200, "class");
   uni_calendar.add(1100, "lecture");
   uni_calendar.add(2400, "all-nighter"); // not added because
                                          // time value is illegal
   ActivityBST sport_calendar;
   sport_calendar.add(930, "walk");
   sport_calendar.add(2100, "walk");
```

```
    cout << "Uni calendar: \n" << uni_calendar << "Sport calendar:
\n" << sport_calendar;
    ActivityBST personal_calendar;
    personal_calendar.add(900, "breakfast");
    personal_calendar.add(2300, "sleep");
    personal_calendar.add(1600, "nap");
    personal_calendar.add(1230, "lunch");
    cout << "Personal calendar: \n" << personal_calendar;
    ActivityBST overall_calendar = sport_calendar +
personal_calendar;
    cout << "Overall calendar: \n" << overall_calendar;
    ActivityBST extended_calendar(uni_calendar);
    extended_calendar.add(1500, "homework");
    cout << "Extended calendar: \n" << extended_calendar;
    extended_calendar+=personal_calendar+= sport_calendar;
    cout << "Personal calendar: \n" << personal_calendar;
    cout << "Extended calendar: \n" << extended_calendar;
    return 0;
}
```

Console output:

```
Uni calendar:
[10:00] - meeting
[11:00] - lecture
[12:00] - class
Sport calendar:
[09:30] - walk
[21:00] - walk
Personal calendar:
[09:00] - breakfast
[12:30] - lunch
[16:00] - nap
[23:00] - sleep
Overall calendar:
[09:00] - breakfast
[09:30] - walk
[12:30] - lunch
[16:00] - nap
[21:00] - walk
[23:00] - sleep
Extended calendar:
[10:00] - meeting
```

```
[11:00] - lecture
[12:00] - class
[15:00] - homework
Personal calendar:
[09:00] - breakfast
[09:30] - walk
[12:30] - lunch
[16:00] - nap
[21:00] - walk
[23:00] - sleep
Extended calendar:
[09:00] - breakfast
[09:30] - walk
[10:00] - meeting
[11:00] - lecture
[12:00] - class
[12:30] - lunch
[15:00] - homework
[16:00] - nap
[21:00] - walk
[23:00] - sleep
```

Sample main program 2

main.cpp

```cpp
#include "ActivityBST.h"
#include <iostream>
using namespace std;

void print_calendar(const ActivityBST& cal){
   cout << "Calender:\n" << cal;
   ActivityBST study_sessions;
   study_sessions.add(1000, "study");
   study_sessions.add(1600, "study");
   cout << "Calender with study sessions:\n" << study_sessions +
cal;
   ActivityBST meetings;
   meetings.add(1300, "meeting");
   meetings.add(900, "meeting");
   cout << "Calender with meetings:\n" << cal + meetings;
   ActivityBST complete_calendar;
   complete_calendar+=cal;
```

```
   complete_calendar+=study_sessions+meetings;
   cout << "Complete calendar:\n" << complete_calendar;
}

int main(){
   ActivityBST calendar;
   calendar.add(800, "breakfast");
   calendar.add(1200, "lunch");
   calendar.add(2000, "dinner");

   print_calendar(calendar);
  return 0;
}
```

Console output:

```
Calender:
[08:00] - breakfast
[12:00] - lunch
[20:00] - dinner
Calender with study sessions:
[08:00] - breakfast
[10:00] - study
[12:00] - lunch
[16:00] - study
[20:00] - dinner
Calender with meetings:
[08:00] - breakfast
[09:00] - meeting
[12:00] - lunch
[13:00] - meeting
[20:00] - dinner
Complete calendar:
[08:00] - breakfast
[09:00] - meeting
[10:00] - study
[12:00] - lunch
[13:00] - meeting
[16:00] - study
[20:00] - dinner
```

Submission rules

In order to get full credit, your programs must be efficient and well presented, the presence of any redundant computation or bad indentation, missing comments, or irrelevant comments are going to decrease your grades. You also have to use understandable identifier names and informative prompts. Modularity is also important; you have to use functions wherever needed and appropriate.

When we grade your homework we pay attention to these issues. Moreover, in order to observe the real performance of your codes, we will run your programs in Release mode and we will test your programs with very large test cases.

What and where to submit (PLEASE READ, IMPORTANT)

You must write your solution in C++. It'd be a good idea to write your name and last name in the program (as a comment line of course). Submission guidelines are below. Some parts of the grading process are automatic. Students are expected to strictly follow these guidelines in order to have a smooth grading process. If you do not follow these guidelines, depending on the severity of the problem created during the grading process, 5 or more penalty points are to be deducted from the grade.

Your submission will consist of only TWO code files:

1. `ActivityBST.h`
2. `ActivityBST.cpp`

Please don't change the names of these files. Submit them with the same names shown above.

You shouldn't add any other files besides your code to the submission folder.

Place these two code files inside a folder named with the following convention:

`SUCourseUserName_YourLastname_YourName_HWnumber`

Your SUCourse user name is your SUNet username that is used for your sabanciuniv e-mails. Do NOT use any spaces, non-ASCII and Turkish characters in the folder name. For example, if your SUCourse username is cago, your first name is Taha Çağlayan, and your last name is Özbugsızkodyazaroğlu, then the folder name must be:

`cago_Ozbugsizkodyazaroglu_TahaCaglayan_1`

Compress this folder using a compression program such as WinZip or WinRAR. Please use "zip" compression. "rar", "7z" or any other compression mechanisms are NOT allowed. Our homework processing system only works with zip files. Therefore, make sure that the resulting compressed file has a zip extension. Check that your compressed file opens up correctly and it contains all your code files. You will receive no credits if your compressed folder does not expand or it does not contain the correct files. The name of the zip file follows the same convention. The zip file for the homework submission by the student mentioned above would be:

`cago_Ozbugsizkodyazaroglu_TahaCaglayan_1`

Submit via SUCourse ONLY! You will receive no credits if you submit by other means (e-mail, paper, etc.). Successful submission is one of the requirements of the homework. If for some reason you cannot successfully submit your homework and we cannot grade it, your grade will be 0.

Good Luck!

Amro