CS205 C/C++ Program Design Assignment 2

Semester: 2022 Fall

Introduction

In computer science, binary search tree (BST) is a sorted data structure, which can achieve dynamic data managing in average time complexity $\mathrm{O}(\log n)$.

In Assignment 2, you will be firstly required to implement some functions about tree nodes and binary search tree. At last, we will introduce a manipulation called *Splay*, which can adjust the structure of the tree when accessing specific element in the tree, bubbling the target to the root of the tree, so that the most frequently-accessed data will be near to the root – which can reduce the access time to the most frequently-used data.

If you are not major in computer science, you can briefly check some key concepts from:

- Tree [Wiki] [Baidu]
- Binary Tree [Wiki] [Baidu]
- Binary Search Tree [Wiki] [OI-Wiki]
- Splay [Wiki] [OI-Wiki]
- Tips: We cannot gurantee the academic correctness and authority of these materials. Please check the definitions from professional textbooks if mind.

These concept is very important for you to finish tasks in this assignment. Please make sure you have had general idea about them before you start.

Tasks

Part 1. Add Node (20 pts)

Part 2. Judge Child Direction (10 pts)

Part 3. Insert into Binary Search Tree (20 pts)

Part 4. Find Element in Binary Search Tree (20 pts)

Part 5. Splay (30 pts)

(You can find an online doc for your tasks via https://cs205-s22.github.io/assign2)

Prelude

For the convenience to correctness checking, we provide you the basic data structure and some macros used for this assignment. You **MUST** use the stucture **tree_node** and structure **BST** provided in `assign2_tree.hpp` to finish this assignment; you **CANNOT** modify the layout of data filed in this structure; or **NO** scores will be given for the code part in Assignment 2.

We also provide another exception structure in `assign2_exception.hpp`. We use 32-bits width data to store different kinds of exceptions, and we define the exception types for different bits in this header file. You shall learn these macros by yourself, and correctly handle exceptions *mentioned in the docs* in your program.

You can briefly check the definition of the structure and macros; and implement the functions you think neccessary in `assign2.cpp` which is created by yourself for Assignment 2.

Part One - Add Node

For a binary tree, one of the most basic operation is adding node. We define function

```
exception add_node(tree_node *father, tree_node *child, int child_direction)
```

as adding node `{child}` to `{father}`'s corresponding child according to the `{child_direction}`. If `{child_direction}` equals `{CHILD_DIRECTION_LEFT}`, add `{child}` to `{father}`'s left child; and if `{child_direction}` equals `{CHILD_DIRECTION_RIGHT}`, add `{child}` to `{father}`'s right child.

Exceptions you **SHOULD** handle:

- `NULL_POINTER_EXCEPTION`
- `DUPLICATED_LEFT_CHILD_EXCEPTION`
- `DUPLICATED_RIGHT_CHILD_EXCEPTION`
- `DUPLICATED_FATHER_EXCEPTION`
- `INVALID_CHILD_DIRECTION_EXCEPTION`

What you **SHOULD** do:

- Implement `exception add_node(tree_node *father, tree_node *child, int child_direction)` in `assign2.cpp`

Part Two - Judge Child Direction

In this part, you are required to implement a function to judge the direction of a child to its father, i.e., whether a child is the left or right child of its father. We define function

```
exception judge_child_direction(tree_node *node, int *child_direction)
```

as judging whether the {node} is the left or right child of its father, and store the result to the address of {child_direction}.

Exceptions you **SHOULD** handle:

- NULL_POINTER_EXCEPTION
- ROOTS_FATHER_EXCEPTION

What you **SHOULD** do:

- Implement exception judge_child_direciton(tree_node *node, int *child_direction) in assign2.cpp

Part Three - Insert into Binary Search Tree

We define function

```
exception insert_into_BST(BST *bst, uint64_t data, tree_node
↪  **inserted_node)
```

as inserting specific {data} into {bst}; and store the inserted tree node to the address of {inserted_node}. Because we use the return value as exception handling, we must use another pointer in parameter to specify where to store the inserted tree node. We use {inserted_node} here, a pointer to {tree_node*} to specify the address, in which the function should store the inserted tree node. If you inserted a *new* node, just store the address of the new node to this address; or if you update an existing node, please store the address of updated node to this address.

Exceptions you **SHOULD** handle:

- NULL_POINTER_EXCEPTION
- NULL_COMP_FUNCTION_EXCEPTION
- All the exceptions should be handled in Part One.

What you **SHOULD** do:

- Implement exception insert_into_BST(BST *bst, uint64_t data, tree_node **inserted_node) in assign2.cpp

Part Four - Find Element in Binary Search Tree

We define function

`exception find_in_BST(BST *bst, uint64_t data, tree_node **target_node)`

as finding specific data in the BST, and storing the target tree node into `{target_node}`.

Exceptions you **SHOULD** handle:

- `NULL_POINTER_EXCEPTION`
- `NULL_COMP_FUNCTION_EXCEPTION`

What you **SHOULD** do:

- Implement `exception find_in_BST(BST *bst, uint64_t data, tree_node **target_node)` in `assign2.cpp`

Part Five - Splay

The target for splay function is to adjust the structure of the BST without changing the in-order traversal, putting the target node to the root. We define function

`exception splay(BST *bst, tree_node *node)`

as *splay* tree node `node` to the root of binary search tree if `node` is in `bst`.

Tips:

- The basic two operations to the node: Zig, Zag
- The three conbination operations to the node according to the structure: Zig-Zig/Zag-Zag, Zig-Zag/Zag-Zig, Zig/Zag
- Reuse the code
- Be aware of **NULL** pointers

Exceptions you **SHOULD** handle:

- `NULL_POINTER_EXCEPTION`
- `NULL_COMP_FUNCTION_EXCEPTION`
- `SPLAY_NODE_NOT_IN_TREE_EXCEPTION`
- All the exceptions should be handled in Part One

What you **SHOULD** do:

- Implement `exception splay(BST *bst, tree_node *node)` in `assign2.cpp`

Option

- `tree_node::data:uint64_t`: The data segment in `tree_node` is defined as `uint64_t`. **But it does not mean that all data stored in this segment is simply explained as 64-bits width integer**. We only save 8-bytes width memory for the data of any representation.
- `tree_node::node_count:uint32_t`: This field is used to record the number of occurrences of a certain data. When inserting the same data, instead of inserting nodes repeatedly, please update this field to count duplicated data.
- `tree_node::tree_count:uint32_t`: This field is used to record the size of the subtree (including the root). Please update this field whenever you change the structure of the BST.
- `BST::*comp:int(uint64_t, uint64_t)`: This is the comparasion function for the data field in specific BST. We specify this function **SHOULD** return 0 when these two data fields equal; return a number less than 0 when the previous data is smaller than the later one; return a number greater than 0 when the previous data is greater than the later one.
- `assign2_exception::*`:
  - `exception`: We use a 32-bits width integer to represent our exceptions. For some single bits, we define some specific exceptions for them. Usage:
    * Record exception into `exception`: `exception |= NULL_POINTER_EXCEPTION`
    * Judge whether `exception` records specific exception: `if (exception & NULL_POINTER_EXCEPTION)`
    * **Attention**: In this assignment, exception is not handled by throw. When you throw a exception, most of time you just throw the most-previous exception. In our assignment, you are required to handle all exception occuring, i.e., in your function, `EXCEPTION1` and `EXCEPTION2` may both occur, then you should contain the information for both of them in your return value.
  - `NULL_POINTER_EXCEPTION`: Possible accessing to member of null pointer occurs, please record this exception.
  - `DUPLICATED_LEFT_CHILD_EXCEPTION`: The user wants to add a new child node to a father's left child, while the father has already had a left child.
  - `DUPLICATED_RIGHT_CHILD_EXCEPTION`: The user wants to add a new child node to a father's right child, while the father has already had a right child.
  - `DUPLICATED_FATHER_EXCEPTION`: The user wants to add a child node to a new father, while the child node has already had a father node.
  - `INVALID_CHILD_DIRECTION_EXCEPTION`: The user uses an invalid value for `child_direction` field.
  - `ROOTS_FATHER_EXCEPTION`: The user wants to access the father node of a root node.
  - `NULL_COMP_FUNCTION_EXCEPTION`: The user wants to insert/find/splay a nodes in

BST, while there is no comp function defined for the BST.

- SPLAY_NODE_NOT_IN_TREE_EXCEPTION: The user wants to splay a node to the root of a BST, while this node does not exist in this BST.