CSCE4133/5133 – Algorithms
Fall 2023

Assignment 3
Binary Search Tree

Out Date: Oct. 06, 2023
Due Date: **Oct. 27, 2023**

Instructions:

- **Written Format & Template:** Students can use either Google Doc or Latex
- Write your full name, email address and student ID in the report.
- Write the number of "Late Days" the student has used in the report.
- Submission via BlackBoard
- **Policy:** Can be solved in individual & review the late-day policy.
- **Submissions:** Your submission should be a PDF for the written section & a zip file with your source code implementation.
- For more information, please visit course website for the homework policy.

1. Overview

In homework 2, we studied the Breadth First Search (BFS) and the Depth First Search (DFS) algorithms. According to the property of BFS, the algorithm always produces the shortest path if the graph is unweighted. In other words, BFS produces the path with least number of edges. However, in practice, each edge of a graph has a weight, e.g., the distance between two buildings. Also, we always want to go to the destination with the shortest distance. In this case, BFS cannot produce the shortest path. Interestingly, we can modify the BFS algorithm to find the shortest path. In particular, at each time, instead of popping out the node from head of the queue, we are going to pop out the node with the shortest path in the current queue. Performing this operator takes $O(n)$ (n is the number of nodes in the queue) to search for the maximum node in the current queue. We can optimize this step using binary search tree and reduce the complexity to $O(\log n)$.

In this homework 3, we are going to implement ***Binary Search Tree*** to store data in order so that we can search or find the minimum value quickly. The map is represented as a graph structure where each node represents one building (or a landmark), and an undirected edge between two nodes represents the path between buildings/landmarks. A list of edges in a graph is represented by an adjacency list that is implemented by a linked list (each edge also includes a weight value). The modified BFS algorithm has been provided.

In this assignment, you are ***REQUIRED*** to implement the ***Binary Search Tree***. For graduate students, you have an additional question which is implementing the ***AVL Tree***.

	Binary Search Tree	**AVL Tree**
Undergrad	Required	Optional
Graduate	Required	Required

If you think the homework is too long, don't worry, it is long because we provide you with the detail of the descriptions. There are some hints that may help to be success in this homework:

- You should start your homework early.
- You should read the descriptions of the homework carefully before you start working on it.
- Before implementing the required functions, you should try to compile the source code first. Basically, the source code with an empty implementation can be compiled successfully. This step will help you to understand the procedure of each problem.
- If you forgot binary search tree, you are encouraged to revise the lecture slides.

The source code is organized as follows:

File or Folder	***Required Implementation***	***Purpose***
`assets/`	No	Contain the data of homework
`include/`	No	Contain headers files
`src/`	No	Contain source code files

`bin/`	No	Contain an executable file
`include/linked_list.hpp`	No	Define the linked list structure
`include/queue.hpp`	No	Define the queue structure
`include/stack.hpp`	No	Define the stack structure
`include/graph.hpp`	No	Define the graph structure
`include/search.hpp`	No	Define the header of BFS
`include/bst.hpp`	No	Define the header of BST
`include/bst.hpp`	No	Define the header of AVL
`src/linked_list.cpp`	No	Implement the linked list structure
`src/queue.cpp`	No	Implement the queue structure
`src/stack.cpp`	No	Implement the stack structure
`src/graph.cpp`	No	Implement the graph structure and algorithms
`src/bfs.cpp`	No	Implement the modified BFS algorithm
`src/bst.cpp`	Yes	Implement the BST
`src/avl.cpp`	Yes	Implement the AVL Tree
`src/main.cpp`	No	Implement the main program
`Makefile`	No	Define compilation rules

2. Implementation

a. Binary Search Tree (BST) (60 points)

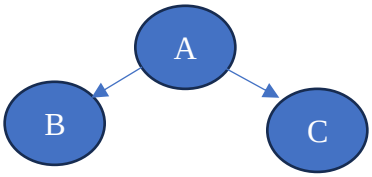
In this section, we are going to implement the basic version of BST. First, we will review the definition of the node in BST and class BST.

```
struct BSTNode {
    int key;
    int height;
    int meta;
    BSTNode *left;
    BSTNode *right;

    BSTNode();
    BSTNode(int key, int height = 0, int meta = 0, BSTNode *left = NULL, BSTNode *right = NULL);
    ~BSTNode();
};

class BST {
    protected:
        BSTNode *root;
    public:
```

```
    BST();
    ~BST();
    BSTNode* find(int key);
    BSTNode* popMaximum();
    BSTNode* popMinimum();
    BSTNode* insert(int key, int meta = 0);
    BSTNode* remove(int key);
};
```

The BSTNode five attributes, i.e. the key of the node, the height of the node, the meta data of the node (you can simply think that is additional information that you want to embed into the node, for example, in the modified BFS, each BST node contains the distance as the key and the vertex index as the meta data), the left and right children pointers of the node. There are three methods, i.e., default constructor, the constructor with given values, and the deconstructor. These methods are already provided.

For class BST, there is a private attribute which is a root of the BST. There are four public methods, i.e. a constructor, a deconstructor, find the given key, pop out the maximum node, pop out the minimum node, insert a new key with the meta data, remove a key. *Although the meta data does not play any specific role in the binary search node and binary search tree, you have to store the meta data into the node.* In our BST, we are ALLOWED to insert duplicated value in the binary search tree. Therefore, in our BST, the rule of each binary search tree node is $node \rightarrow left \rightarrow key \leq node \rightarrow key < right \rightarrow key$.



$B \leq A < C$

The constructor of BST initializes the empty BST, the implementation of the constructor is provided.

```
BST::BST() {
    this->root = NULL;
}
```

The deconstructor of BST will deallocate memory of all nodes in the BST. ***You have to implement the deconstructor to release memory of all nodes.***

```
BST::~BST() {
    // YOUR CODE HERE
}
```

The find method will find the node that contains the key. *You have to implement the find method to return the node containing the given key.*

```
BSTNode* BST::find(int key) {
    // YOUR CODE HERE
}
```

The pop maximum method will return the node that contains the maximum value and remove that node out BST but ***DO NOT release the memory of this maximum node.*** *You have to implement the pop maximum method to return the node containing the maximum value and remove it out of the BST.*

```
BSTNode* BST::popMaximum() {
    // YOUR CODE HERE
}
```

The pop minimum method will return the node that contains the minimum value and remove that node out BST but ***DO NOT release the memory of this minimum node.*** *You have to implement the pop minium method to return the node containing the minum value and remove it out of the BST.*

```
BSTNode* BST::popMinimum() {
    // YOUR CODE HERE
}
```

The insert method will insert a new node containing the key and meta data. *You have to implement the insert method to insert a new node with a given key and meta data into BST and return the new root of BST.*

```
BSTNode* BST::insert(int key, int meta) {
    // YOUR CODE HERE
}
```

The remove method will remove a new node containing the key and meta data. *You have to implement the remove method to remove a node containing the given key out of BST and return the new root of BST.*

```
BSTNode* BST::remove(int key) {
    // YOUR CODE HERE
}
```

You are ALLOWED to define or implement any additional functions to support your BST implementation if you think it is necessary. In this section, you do not need to care the height of the tree. The attribute height in the `BSTNode` is used to implement the self-balanced binary search tree for in the next section.

b. Self-balanced Binary Search Tree: AVL Tree (20 points, bonus)

In this section, we are going to implement the AVL Tree. Since the AVL tree is also a binary search tree, class AVL inherits from class BST.

```
class AVL: public BST {
   public:
      AVL();
      ~AVL();

      int balanceFactor(BSTNode *node);
      void update(BSTNode *node);
      BSTNode* rotateRight(BSTNode *node);
      BSTNode* rotateLeft(BSTNode *node);
      BSTNode* balance(BSTNode *node);

      BSTNode* insertHelper(int key, int meta, BSTNode *node);
      BSTNode* insert(int key, int meta = 0);

      BSTNode* findMinimum(BSTNode *node);
      BSTNode* removeMinimum(BSTNode *node);
      BSTNode* removeHelper(BSTNode *node, int key);
      BSTNode* remove(int key);
};
```

There are constructors and deconstructor to initialize and deallocate the AVL tree. There are several private methods that you will need to implement. In this question, we do not need to implement the find method since it will use the find method of the binary search tree since AVL is also a BST, thus, the searching method remain the same.

The constructor of AVL initializes the empty AVL, the implementation of the constructor is provided.

```
AVL::AVL() : BST() {
   // The AVL constructor will initialize the empty AVL tree via the BST constructor
}
```

The deconstructor of AVL will deallocate memory of all nodes in the BST. ***You do not need to implement this method since since it will use the deconstructor of BST.***

```
AVL::~AVL() {
```

```
    // The AVL deconstructor will deallocate memory of all nodes in AVL via the BST deconstructor
}
```

The balance factor method will calculate the height difference between right and left children nodes of the given node, i.e., ***height of right node – height of left node***. *You need to implement the balance factor method to return the height difference between right and left nodes. We assume that the NULL pointer has the height of 0.*

```
int AVL::balanceFactor(BSTNode *node) {
    // YOUR CODE HERE
}
```

The update method will update the height of the given node, i.e., ***height of the given node = max(height of left node, height of right node) + 1***. *You need to implement the update method to update the height of the given node. We assume that the NULL pointer has the height of 0.*

```
void AVL::update(BSTNode *node) {
    // YOUR CODE HERE
}
```

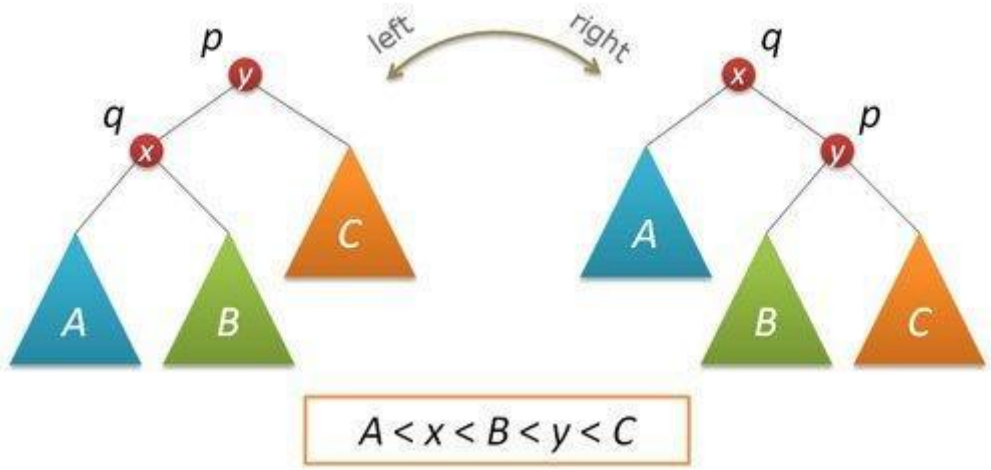
The rotate right method will rotate a given node to the right. *You need to implement the rotate right method to rotate the node into the right and return the new root of the subtree.*

```
BSTNode* AVL::rotateRight(BSTNode *node) {
    // YOUR CODE HERE
}
```

The rotate left method will rotate a given node to the left. *You need to implement the rotate left method to rotate the node into the left and return the new root of the subtree.*

```
BSTNode* AVL::rotateLeft(BSTNode *node) {
    // YOUR CODE HERE
}
```

The following figure illustrates the rotate left and right methods. When we rotate p to the right, q will be the new root of the subtree. When we rotate q to the left, p will be the new root of the subtree.



The balance method will balance the current subtree that satisfy the rules of AVL. ***You have to implement the balance method to balance the current subtree where node is the root of the subtree.***

```
BSTNode *AVL::balance(BSTNode *node) {
    // YOUR CODE HERE
}
```

The insert helper method is the support method that will help to insert a given key and meta to the current AVL subtree where node is the root of the subtree. ***This one is optional.***

```
BSTNode* AVL::insertHelper(int key, int meta, BSTNode *node) {
    // YOU CAN IGNORE THIS FUNCTION IF YOU DO NOT WANT TO THIS FUNCTION
    // YOUR CODE HERE
}
```

The insert method will insert a new node with a given key and meta data. ***You have to implement this method to insert a new node into AVL and return the new root of AVL. However, if you implement the insertHelper function, this function can simply call the insertHelper, i.e., return this->root = this->insertHelper(key, meta, this->root)***

```
BSTNode *AVL::insert(int key, int meta) {
    // YOUR CODE HERE
    //return this->root = this->insertHelper(key, meta, this->root); // Uncomment this one if you implement the insertHelper to insert a node into AVL.
}
```

The find minimum method will find the node containing the minimum value in the current subtree of node. This method is used to support the remove method, if you will not use this method in your remove method, you can ignore this one.

```
BSTNode* AVL::findMinimum(BSTNode *node) {
    // YOU CAN IGNORE THIS FUNCTION IF YOU DO NOT WANT TO THIS FUNCTION
    // YOUR CODE HERE
}
```

The remove minimum method will remove the node containing the minimum value in the current subtree of node. This method is used to support the remove method, if you will not use this method in your remove method, you can ignore this one.

```
BSTNode* AVL::removeMinimum(BSTNode *node) {
    // YOU CAN IGNORE THIS FUNCTION IF YOU DO NOT WANT TO THIS FUNCTION
    // YOUR CODE HERE
}
```

The remove helper method will remove the node containing the given key in the current subtree of node. This method is used to support the remove method, if you will not use this method in your remove method, you can ignore this one.

```
BSTNode* AVL::removeHelper(BSTNode *node, int key) {
    // YOU CAN IGNORE THIS FUNCTION IF YOU DO NOT WANT TO THIS FUNCTION
    // YOUR CODE HERE
}
```

The remove method will remove a node containing the given key. *You have to implement this method to remove the node containing the key and return the new root of AVL. However, if you implement the removeHelper function, this function can simply call the removeHelper, i.e., return this->root = this->removeHelper(this->root, key).*

```
BSTNode *AVL::remove(int key) {
    // YOUR CODE HERE
    // return this->root = this->removeHelper(this->root, key); // Uncomment this one if you implement the removeHelper to remove a node out of AVL.
}
```

You are ALLOWED to define or implement any additional functions to support your AVL implementation if you think it is necessary.

3. Compilation and Testing (20 points)

In this homework 3, we use the same environment (Make, GCC/G++, OpenCV Library) used in the previous ones to compile the source code.

In this homework, ***you use only a single Makefile***. It will automatically detect your Operator System (e.g., Linux, MacOS, or Windows) and configurate the compilation rules. For Windows users, you have to make sure that you set the correct paths for MINGW_BIN and OPENCV_DIR:
MINGW_BIN=<path to mingw64>/bin
OPENCV_DIR=<path to opencv>/build/install
For example, if we stored mingw64 and opencv in C:/, we edit these paths as follows:
MINGW_BIN=C:/mingw64/bin
OPENCV_DIR=C:/opencv/build/install

OpenCV Library: In our homework, OpenCV is OPTIONAL and only used for the visualization purpose, you can work on your homework WITHOUT installing and using OpenCV. If you do not want to use OpenCV for visualization, open file ***Makefile*** and edit line ***OPENCV=1*** to ***OPENCV=0.***

If you are graduate students, please edit line GRADUATE=0 to GRADUATE=1 in your Makefile. If you are undergrad students, you can ignore this one. However, if you want to implement AVL, you can set GRADUATE=1 in your Makefile.

To compile and test your source code, you open the terminal and go to the source code folder and type the following commands: **make (**or **mingw32-make** on Windows**).** If your implementation is correct, you should pass all the unit tests.

For Windows users, if you would like to compile with OpenCV, you have to copy the following files from ***C:\opencv-3.4.13\build\bin*** to your Homework folder:

- libopencv_core3413.dll
- libopencv_highgui3413.dll
- libopencv_imgcodecs3413.dll
- libopencv_imgproc3413.dll