Homework #3

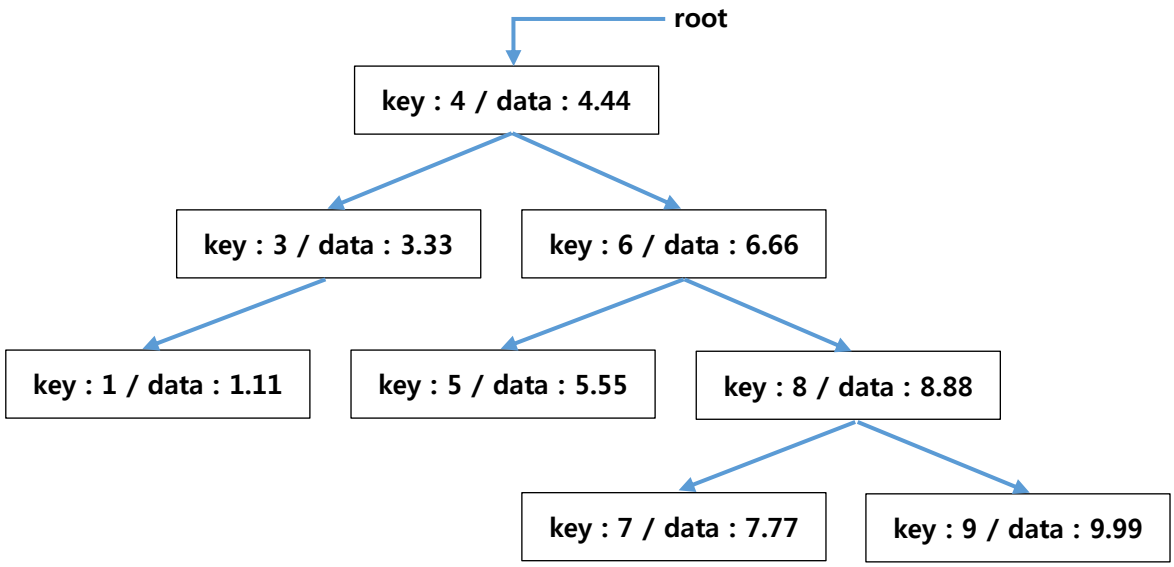
Nov. 11th (Mon.) ~ Nov. 25th (Mon.) 23:59

*** Be sure to read the note on the last page.**

P1 (20pt). [BST] Implement preorder, postorder and levelorder function for Dictionary using Binary Search Tree to satisfy the following conditions. Please refer to HW3_BST.h class which is implemented in the lab class. (HW2_Queue.h can be used if needed)

Files to submit : HW3_BST.h, HW3_BST.cpp (**※ Don't include main function in .cpp file**)

Example of BST Dictionary is as below.



TreeNode Class

- **Definition**
  - ■ Same as the *TreeNode* class as defined in Lab 3

BST Class

- **Definition**
  - ■ Same as the *BST* class as defined in Lab 3

- **Member Function** (define as **Protected**)
  - ■ You can use all protected functions provided (same as defined in Lab 3)

- **void preorder(TreeNode *curr_node)**
  - Do preorder traversal from *curr_node* and print the result of it.
    1. Print the *key* and the *data* of *curr_node* (**<key, data>**)
    2. Do preorder traversal with the left sub-tree of *curr_node*
    3. Do preorder traversal with the right sub-tree of *curr_node*
- **void postorder(TreeNode *curr_node)**
  - Do postorder traversal form *curr_node* and print the result of it.
    1. Do postorder traversal with the left sub-tree of *curr_node*
    2. Do postorder traversal with the right sub-tree of *curr_node*
    3. Print the *key* and the *data* of *curr_node* (**<key, data>**)
- **void levelorder(TreeNode *curr_node)**
  - Print the traversal result from *curr_node* to nodes of its sub-tree in increasing order by level
  - You can use HW2_Queue.h if needed
- **void merge(TreeNode *curr_node, TreeNode *merge_node)**
  - Merge the sub-tree whose root is *merge_node* into the sub-tree whose root is *curr_node*
  - The sub-tree whose root is *curr_node* should keep the property of BST
  - If both sub-tree have the node whose *key* is the same, replace *data* of *curr_node* tree to that of *merge_node* tree
  - Do not modify the tree rooted from *merge_node*

- **Member Function** (define as **Public**)
  - **void preorder()**
    - Print the nodes of BST by preorder traversal
  - **void postorder()**
    - Print the nodes of BST by postorder traversal
  - **void levelorder()**
    - Print the nodes of BST by levelorder traversal
  - **void merge(const BST& bst)**
    - Merge *bst* into this BST using protected **merge()**
    - Do not modify *bst*

E.g.) Traversal results for each case in HW3_BST_Test.cpp:

Preorder Traversal from Root :

[4, 4.44] [3, 3.33] [1, 1.11] [6, 6.66] [5, 5.55] [8, 8.88] [7, 7.77] [9, 9.99]

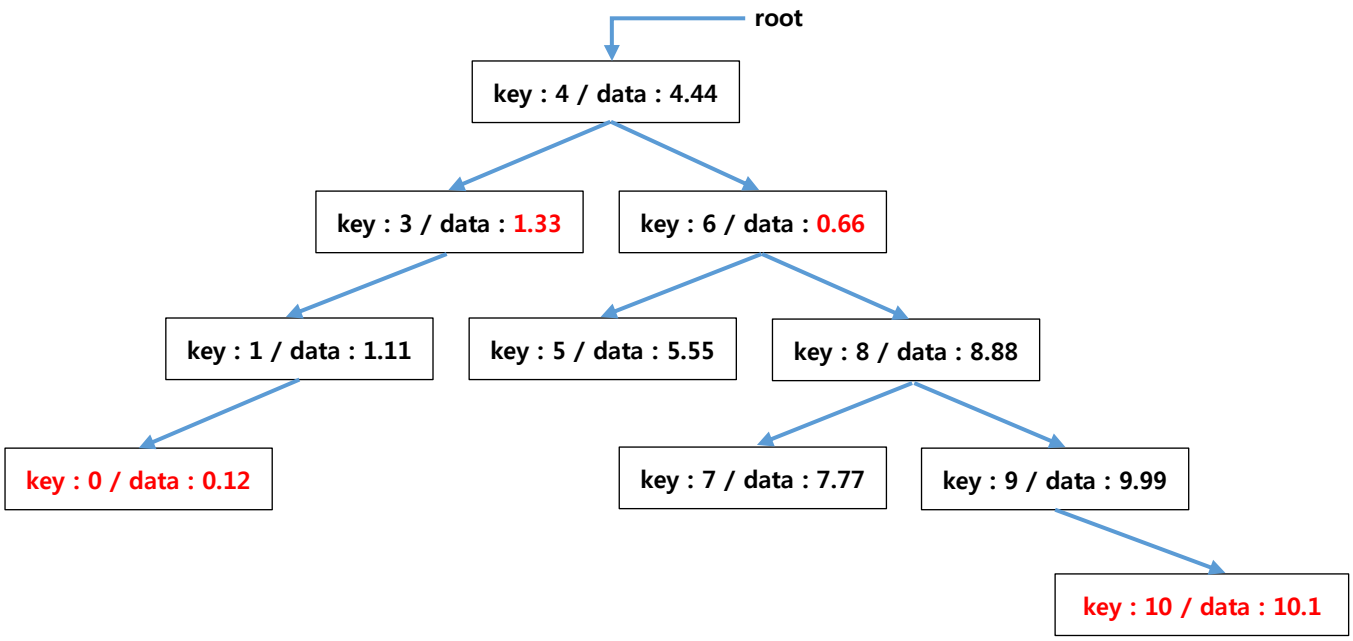
Postorder Traversal from Root :

[1, 1.11] [3, 3.33] [5, 5.55] [7, 7.77] [9, 9.99] [8, 8.88] [6, 6.66] [4, 4.44]

Levelorder Traversal from Root :

[4, 4.44] [3, 3.33] [6, 6.66] [1, 1.11] [5, 5.55] [8, 8.88] [7, 7.77] [9, 9.99]

Merge result (dict) in HW3_BST_Test.cpp:

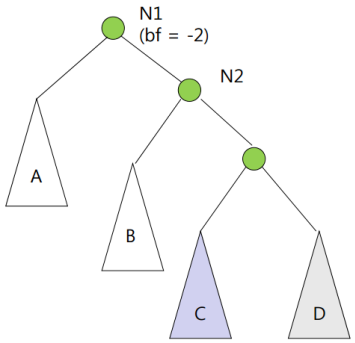


P2 (50pt). [AVL Tree] Implement AVL Class to satisfy the following conditions, using Binary Search Tree which is implemented in P1.

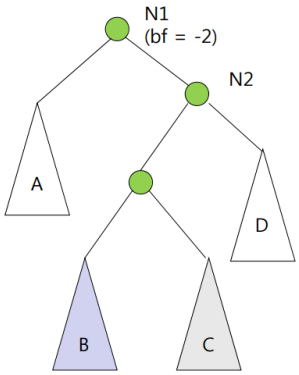
Files to submit : HW3_AVL.h, HW3_AVL.cpp (**※ Don't include main function in .cpp file**)

AVL Class

- **Definition**
  - Class representing height balanced BST
  - Inherited from BST class as public **(class AVL : public BST{})**
- **Constructor** (define as **Public**)
  - **AVL()**
    - Call the constructor of BST to initialize *root* as nullptr
- **Member Function** (define as **Private**)
  - **int BF(TreeNode * curr_node)**
    - Return balanced factor of *curr_node*
  - **void insert(TreeNode *& curr_node, const int& key, const double& data)**
    - Add new node which has *key* and *data* into the sub-tree whose *root* is *curr_node*
    - If there is node which has the same *key*, replace data of the node with *data*
    - After insert, the following property should be satisfied
      - *Key* of left child node should be smaller than that of the parent node
      - *Key* of right child node should be greater than that of the parent node
    - Calculate imbalance of sub-tree using **BF()** and rotate it to make height balanced tree using **rotateSingle()**, **rotateDouble()**
  - **void rotateSingle(TreeNode *& curr_node)**
    - In the imbalance situation, for example, shown in the right figure, do rotation which makes height balanced BST by making **N2** into *root*
    - Refer to the lecture materials



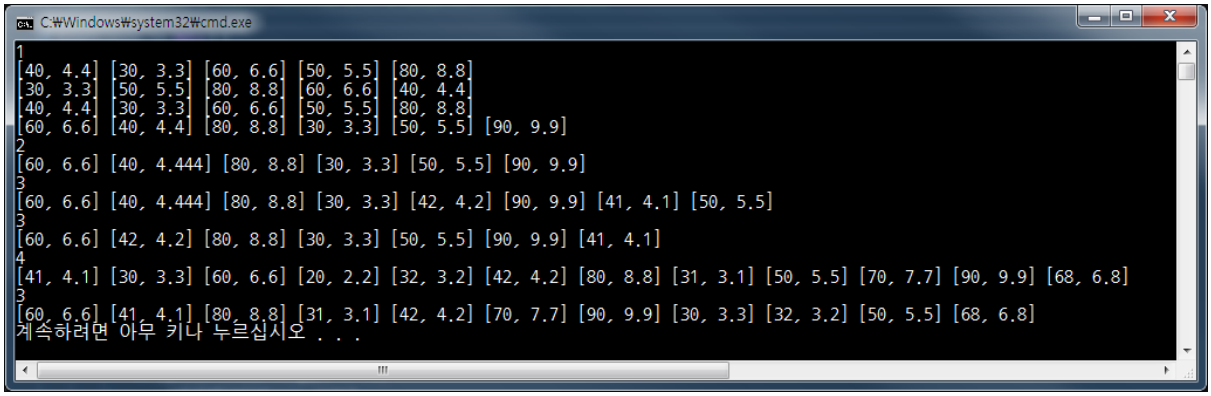
- **void rotateDouble(TreeNode *& curr_node)**
  - In the imbalance situation, for example, shown in the right figure, do rotation which makes height balanced BST by making the left child node of **N2** into *root*
  - Refer to the lecture materials



- **void remove(TreeNode *& curr_node, const int& key)**
  - In the sub-tree whose root is curr_node, remove node whose key is *key*
  - You can use **doRemoval()**in the BST Class or define additional member function for deletion
  - First step is similar to **remove()** function in the BST Class
  - After remove, propagated rotation procedure is needed to keep the property of height balanced BST
  - If both **rotateSingle()** and **rotateDouble()** can be done in propagation stage, do **rotateSingle()** first
  - Refer to the lecture materials

- **Member Function** (define as **Public**)
  - **void insert(const int& key, const double& data)**
    - Add new node whose key is *key* and data is *data* in the AVL tree
    - If there is node which has the same *key*, replace data of the node with *data*
  - **void remove(const int& key)**
    - Remove node whose key is *key*

When HW3_AVL_Test.cpp is executed, the result is as follow.



P3 (30pt). [Hashing] We want to make hash table which store data by using integer key. If there is collision between keys, we want to use linear-probe based hashing. Implement HashTable Class to satisfy the following conditions.

Files to submit : HW3_HashTable.h (**※ Implement all functions in the header file**)

HashTable Class

- **Definition**
  - Hash table that stores data using integer key
- **Member Variables** (define as **Private**)
  - **HashNode class**
    - Represent the node stored in hash table
    - Integer *key* and template V class *value* is stored in **HashNode**
  - **HashNode<V> **table** : Hash table storing **HashNode**
  - **HashNode<V> *dummy** : Dummy node for node deletion
  - **int capacity** : The number of **HashNode** that hash table can store
  - **int number** : The number of **HashNode** that hash table stores
- **Constructor** (define as **Public**)
  - **HashTable (int cap)**
    - Initialize *capacity* to *cap* and *number* to 0
    - Create *dummy* whose key is -1 and value is NULL by using the constructor of HashNode class
    - Create array in the *table* to store **HashNode<V>*** and initialize each **HashNode<V>*** to nullptr. It can store **HashNode<V>*** up to *capacity*
- **Destructor** (define as **Public**)
  - **~HashTable()** : delete table, **HashNode** in the *table* and *dummy*
- **Member Functions** (define as **Public**)
  - **int hashFunction (int key)**
    - Hash functions which change *key* into the address of hash table
    - It has been defined as h(*key*) = *key* % *capacity*
  - **void tableDoubling()**
    - If the number of HashNode is greater than the half of the *capacity*, double its *capacity* to avoid performance degradation

      (E.g. If *capacity* is 7 → doubling when 4th node is inserted / If capacity is 8 →

doubling when 5th node is inserted)

- ◆ It is called in **insertNode()** and just double *capacity* of hash table
- ◆ Create new table which has double *capacity*
- ◆ For already stored **HashNode**, do re-hashing for the new *table* (**Do not just copy array!**)
- ◆ Delete original table

■ **void insertNode(int key, V value)**

- ◆ Add new **HashNode** with *key*, *value* in the hash table, where the address of HashNode is from **hashFunction()**
- ◆ If the other node has already occupied the address, add new node in empty address or in the address of dummy using linear-probing
- ◆ If already occupying node has the same *key*, replace its value into *value*
- ◆ Update *number* and if the *number* is greater than the half of *capacity*, call **tableDoubling()**

■ **V deleteNode(int key)**

- ◆ Delete **HashNode** with *key* in hash table and update *number*
- ◆ Check hash table with the address using **hashFunction()** and if there is a node with different key, check the next address due to linear-probing
- ◆ If there is the node with *key*, delete it, store *dummy* and return the *value* of the deleted node
- ◆ If the node with *key* doesn't exist in the hash table, print "key <*key*> does not exist." and return NULL

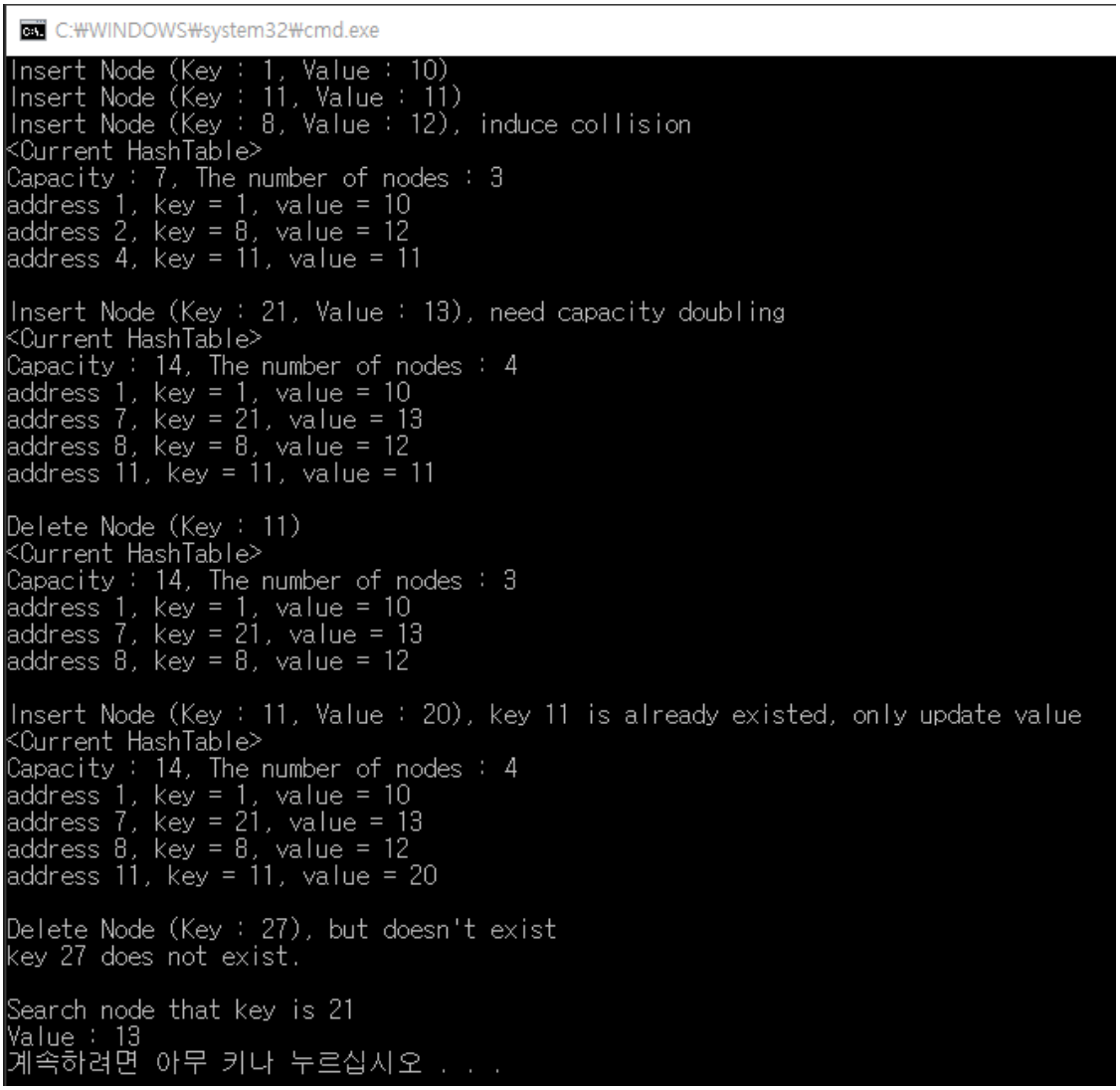
■ **V search(int key)**

- ◆ Check hash table with the address using **hashFunction()**
- ◆ If there is the node with *key*, return the *value* of the node
- ◆ If the node with *key* doesn't exist in the hash table, return NULL

■ **void display()**

- ◆ Print *capacity* and *number* of hash table
- ◆ Print *key*, *value* of HashNode in hash table
- ◆ Refer to the output form in the next page (**must be the same**)

When HW3_HashTable_Test.cpp is executed, the result is as follow.



※ The reason *dummy* is needed : HashNodes which has same address are allocated consecutively by linear-probing. If hash table has nullptr in the address when a node is removed, the nodes after it cannot be found which stored by linear-probing. So dummy should be inserted.

- **Note**
  - **Scoring will be based on execution results in Microsoft Visual Studio 2019.**
  - **Do NOT modify given names of functions and variables! Otherwise score will be deducted.**
  - **You can add member functions or re-define inherited function (virtual keyword in the base class is needed) if needed.**
  - **Submit the files with the exact file names given! Otherwise score will be deducted.**
  - **Output should be in the same form as the example given! Otherwise score will be deducted.**
  - **Scoring will be done with more complex case than the example test case given.**
  - **No plagiarism! If plagiarism is detected, 0 points will be given for the assignment and it will be notified to the professor.**
  - **If you have any question, ask questions via e-mail only if they are not resolved after sufficient search has been done.**
- **How to submit**
  - Write code for each problem in each .h / .cpp file.
  - Files should be saved **with exact file names** given in each problem.
  - Compress your code into **one compressed file**.
    - The compressed file name should be "**HW3_(name)_(student ID).zip**" using zip compression.
    - e.g.) "**HW3_김태환_2017-11111.zip**"
  - Submit the compressed file to the "Assignment 3" on the eTL course page.
- **Deadline for submission**
  - **By Monday, November 25th, 11:59 pm.**
  - Submit the assignment **via e-mail within the deadline** if there is any problem when you submit it on eTL.
    E-mail : ds@snucad.snu.ac.kr
  - **No delay submission (both eTL / e-mail). 0 points for late submission or no submission.**
  - **Make sure that the file is attached and submitted! 0 points for submission without attachment.**