BKU Tree Implementation

- ## Outcome

After successfully completing this project, I will be able to:

- Perform competently tree data structure in programming languages C / C ++.
- Understand and manipulate well two types of binary search tree, AVL and Splay Tree.

- ## Introduction

Binary search tree is a group of storage-capable trees that search for elements with average complexity $O(n \log n)$ which N is the number of stored data points. In which, AVL belongs to a tree group that is balanced with the complexity when searching in all cases is $O(n \log n)$ thanks to the height adjustment operators so that each node satisfies the property of the disparity of the subtrees.

However, if there are data points that are frequently searched and are always at the end of the AVL tree, then using the AVL structure for storage would be inefficient. The Splay Tree was invented to solve the above problem by automatically bringing frequently searched nodes closer to the root node, making searching for these data points more efficient.

The biggest disadvantage of Splay Tree is the unbalanced height, so if the search above is random, the searching on Splay Tree is ineffective due to continuous tree adjustment operations. If an application requests at different stages, the sequence of lookup values will be randomized or concentrated on a few data points, both of these trees are at a disadvantage in different stages.

Therefore, the BKU Tree data structure was given in this assignment as an idea to address the time inefficiency for both trees.

- ## BKU Tree Binary search tree

The BKU Tree Binary search tree is a tree which carries both the AVL tree and the Splay tree in it so that the program can both search like on an AVL Tree, and to search like on a Splay

Tree. Specifically, in the structure of BKU Tree, it will be stored as follows:

1. Splay Tree
2. AVL Tree
3. A queue stores most recent search keys (Maximum number of queue elements was initial ized by the user)

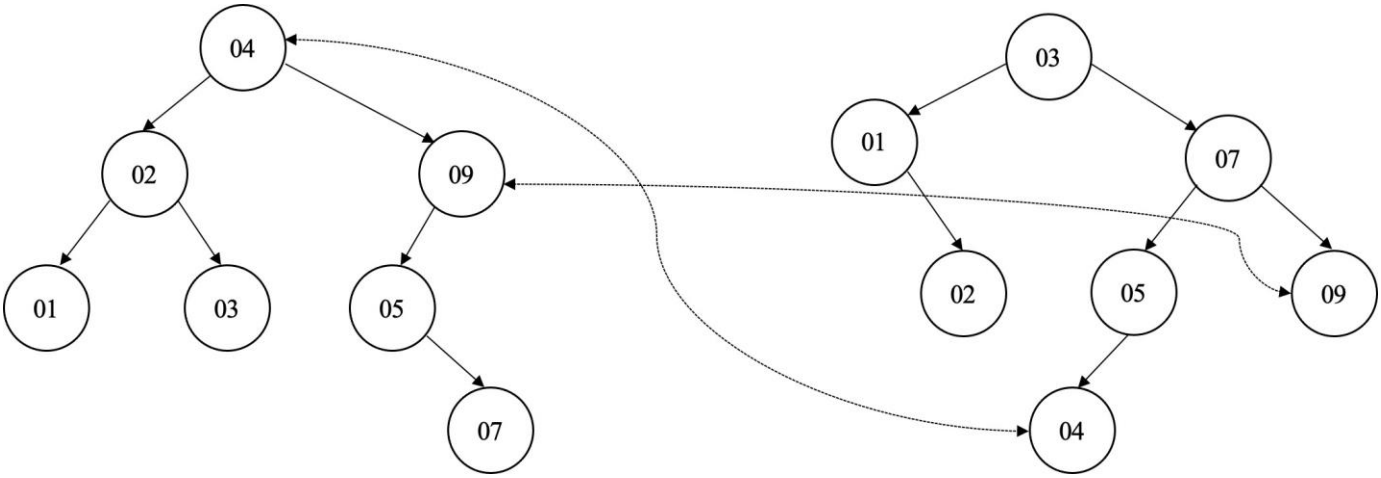
In it, each node in the Splay tree stores the address of (or has a pointer to) the corresponding node in the AVL tree and vice versa.

- **Insertion and deletion operation**

When perform insertion and deletion operations on BKU Tree, the corresponding operation must take place on both the Splay tree and the AVL tree.

- In case of insertion: The added key must be added to the queue. If the queue is already full then we have to delete element at the head of queue (dequeue) before adding.
- In case of deletion: If this key has existed in the queue, we have to delete this key from the queue and add the key currently located at the root of the Splay tree (after deletion).

Example: Insert a list of integer keys [1, 3, 5, 7, 9, 2, 4] into a BKU Tree (the queue size is 5) respectively, we have the following tree in model:



Hình 1: The result after inserting [1, 3, 5, 7, 9, 2, 4] into BKU Tree

Note: To ignore intricacy, there are some corresponding link between nodes in two trees, so we need to know other links to connect two nodes.

- **Search operation**

1. If the key is at the root of the Splay Tree, we return the found node and finish.

2. If the key is in a queue, we search on the Splay Tree. Perform splay one at found node (no recursion).

3. If the key is not in a queue, we do the following step:

(a) From the root node r of the Splay Tree, we refer to the corresponding r' node on the AVL Tree.

(b) Search on the AVL sub-tree where node r0 is the root node.

(c) If found in the AVL tree in step (b), returns the found node f0 .

(d) If not found, we search from the root of the AVL tree total. During the search process, if we go through node r0 , then returns the result is not found. On the contrary, returns the found node f'.

(e) From node f0 of the AVL Tree, we refer to corresponding node f on the Splay Tree. Perform splay once at the node f to bring the node closer to the root node (no recursion).

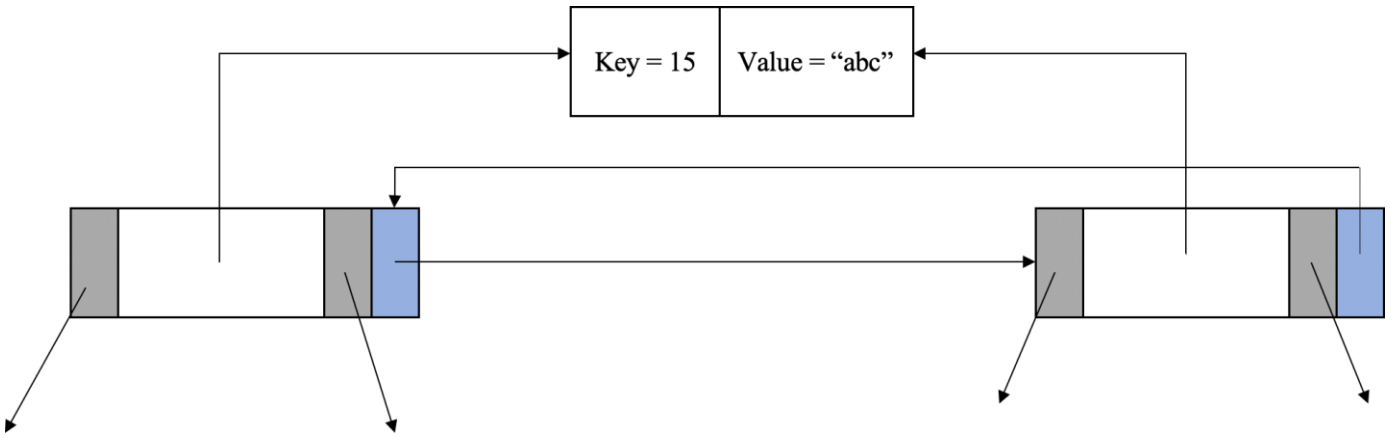
4. Put the key you searched for at the end of the queue (enqueue) (even if it already exists in the queue). In the case of an overloaded queue, we remove the key at the head of queue (dequeue) before adding this key.

- ## File Structure:

Build a program in C++ to implement the BKU Tree data structure proposed above with file structure as follows:

- Class BKUTree(with template Kand template V) represents the BKUTree data structure with attributes and methods:
  - Attributes:
    - AVLTree* avl used to store AVL Tree.
    - SplayTree* splay used to store Splay Tree.
    - queue<K>* keys used to store the recently searched key queue with queueis queue data structure in STL C++.
  - Methods:
    - BKUTree(int maxNumOfKeys = 5) : constructor to set the maximum number of keys that can be stored in the queue. If the user does not pass in, the maximum number of keys to be used is 5.
    - void add(K key,V value) : adds an element with the key and value to the BKU Tree structure. If the key already exists, the program will throw an excep- tion "Duplicate key".
    - void remove(Kkey) : delete an element that has the key from the BKU Tree data structure. If the key does not exist, the program will throw an exception "Not found".
    - V search(K key,vector<K>& traversedList) : searches for an element that has the key in the BKU Tree data structure. If the key does not exist, the program will throw an exception "Not found", otherwise it returns the value corresponding to that key. In addition, traversedList will store the passed keys in turn before finding the element that has the key or until the end in the case of not finding it.
    - traverseNLROnAVL and traverseNLROnSplay: are two methods used to traverse pre-order on the two AVL trees and the Splay tree that BKUTree is storing with func is the pointer function that stores the processing at each traversing each node.

- Nested class Entry within BKU Tree is a structure used to store the key and corresponding value of a data point.
- Nested class AVLTree within BKU Tree is a structure used to store AVL Tree with class Node used to store a node in the AVL Tree including the information: address of the stored entry, the address of the child on the left and on the right, the node's balance factor and
  the address of the corresponding node on the Splay Tree.
- Nested class SplayTree within BKU Tree is a structure used to store Splay Tree with class Node used to store a node in the Splay Tree including the information: address of the stored entry, the address of the child on the left and on the right, the node's balance factor and the address of the corresponding node on the AVL Tree.
- The methods of AVLTree and SplayTree have the same input and output:
  - void add(K key,V value): used to add a Node to which the entry points contains key and value. If the key already exists, a "Duplicate key" exception is thrown.
  - void add(Entry*entry): used to add a node to which the entry points is the entry being passed. If this entry already exists in any node of the tree, a "Duplicate key" exception is thrown.
  - void remove(Kkey) : used to remove a node from the tree. In case the node has key that does not exist, the exception "Not found" is thrown.
  - V search(K key) : search and return the corresponding value of the key-bearing node. In case the node has key that does not exist, the exception "Not found" is thrown.
  - void traverseNLR(void(*func)(K key,V value)) : used to traverse pre-order on the tree with the traversing operation defined by the function pointer func.
  - void clear() : used to clear and release all memory allocated to the tree.



Hình 2: Example for sharing entry between corresponding nodes on two trees

In figure 2, at each node, the grey area point out two subtree on left and right sides, blue one stores the link with corresponding node and the white area keeps the memory of stored entry.

- # Evaluation

Testcases are divided into 3 groups:

1. Testcase 00 - 24: Test your implementations on AVL Tree.
2. Testcase 25 - 59: Test your implementations on Splay Tree.
3. Testcase 60 - 99: Test your implementations on BKU Tree.

On the other hand, the program needs to release whole allocated memory after running. Testcase from 80 to 99 will be test this requirement.

————————THE END————————