Ceng213 - Data Structures
Programming Assignment 2
TreeMap Implementation via Scapegoat Trees

Fall 2022

1 Objectives

In this programming assignment, you are first expected to implement a *scapegoat tree* data structure (i.e., a *self-balancing binary search tree*), in which each node will contain the element and two pointers to the root nodes of its left and right subtrees. The scapegoat tree data structure will include a single pointer that points to the root node of the scapegoat tree. The details of the structure are explained further in the following sections. Then, you will use this specialized scapegoat tree structure to implement a tree map data structure (i.e., a scapegoat tree based map implementation that keeps its entries (key-value pairs) sorted according to the natural ordering of its keys.).

Keywords: *C++, Data Structures, Binary Search Trees, Self-Balancing, Scapegoat Trees, TreeMap Implementation*

2 Scapegoat Tree Implementation (70 pts) [1]

In computer science, a scapegoat tree is a self-balancing binary search tree. It provides worst-case $O(log\ n)$ lookup time (with n as the number of nodes) and $O(log\ n)$ amortized insertion and deletion time. Instead of the small incremental rebalancing operations used by most balanced tree algorithms, scapegoat trees rarely but expensively choose a "scapegoat" and completely rebuild the subtree rooted at the scapegoat ~~into a complete binary tree~~ when the tree becomes unbalanced. Thus, scapegoat trees have $O(n)$ worst-case update performance.

Two conditions must be satisfied at all times for scapegoat trees ((1) is called the *upper bound condition* and (2) is called the *height condition*):

$$\frac{upper\_bound}{2} <= number\_of\_nodes <= upper\_bound \tag{1}$$

$$height\_of\_tree <= log_{3/2}(upper\_bound) \tag{2}$$

In order to make sure that conditions (1) and (2) are satisfied at all times, there are two things to do while ~~doing~~ operating on the scapegoat tree:

- If insertion violates condition (2), the height condition, find a scapegoat node and then rebuild the subtree rooted at that scapegoat node.

[1] Some of the information about scapegoat trees in this section is from the Wikipedia page "Scapegoat tree".

- If removal violates condition (1), the upper bound condition, rebuild the entire tree.

The scapegoat tree data structure used in this assignment is implemented as the class template `ScapegoatTree` with the template argument `T`, which is used as the type of the element stored in the nodes. The node of the scapegoat tree is implemented as the class template `Node` with the template argument `T`, which is the type of the element stored in nodes. `Node` class is the basic building block of the `ScapegoatTree` class. `ScapegoatTree` class has a `Node` pointer (namely `root`) which points to the root node of the scapegoat tree, and an integer (namely `upperBound`) which keeps the current upper bound of the scapegoat tree in its private data field.

The `ScapegoatTree` class has its definition and implementation in *ScapegoatTree.h* file and the `Node` class has its in *Node.h* file.

2.1 `Node` Class

`Node` class represents nodes that constitute scapegoat trees. A `Node` keeps two pointers (namely `left` and `right`) to the root nodes of its left and right subtrees, and a data variable of type `T` (namely `element`) to hold the data. The class has two constructors and the overloaded output operator. They are already implemented for you. You should not change anything in file *Node.h*.

2.2 `ScapegoatTree` Class

`ScapegoatTree` class implements a scapegoat tree data structure ~~with the `root` pointer~~. Previously, data members of `ScapegoatTree` class have been briefly described. Their use will be elaborated in the context of utility functions discussed in the following subsections. You must provide implementations for the following public interface methods that have been declared under indicated portions of *ScapegoatTree.h* file.

2.2.1 `ScapegoatTree();`

This is the default constructor. You should make necessary initializations in this function.

2.2.2 `ScapegoatTree(const ScapegoatTree<T> &obj);`

This is the copy constructor. You should make necessary initializations, create new nodes by copying the nodes of the given `obj`, and insert those new nodes into this scapegoat tree. The structure among the nodes of the given `obj` should also be copied to this scapegoat tree.

2.2.3 `~ScapegoatTree();`

This is the destructor. You should deallocate all the memory that you were allocated before.

2.2.4 `bool isEmpty() const;`

This function should return `true` if this scapegoat tree is empty (i.e., there exists no nodes in this scapegoat tree). If this scapegoat tree is not empty, it should return `false`.

2.2.5 `int getHeight() const;`

This function should return an integer that is the largest number of edges from the root to the most distant leaf node in this scapegoat tree (i.e., the height of this scapegoat tree).

2.2.6 `int getSize() const;`

This function should return an integer that is the number of nodes in this scapegoat tree (i.e., the size of this scapegoat tree).

2.2.7 `bool insert(const T &element);`

You should create a new node with the given `element` and insert it at the appropriate location in this scapegoat tree and return `true`. Don't forget to make necessary pointer modifications in the tree. If there already exists a node with the given `element` in this scapegoat tree, this function should do nothing but return `false`. ~~***You will not be asked to insert duplicated elements into the binary search tree.***~~

After inserting the new node at the appropriate location with the help of ordering property of binary search trees, first you should increment the upper bound, and then you should check the height condition (2) of the scapegoat tree. If the height condition is not violated, you should do nothing. If it is violated, you should find a scapegoat and rebuild the subtree rooted at the scapegoat.

To do so, you should start from the newly inserted node and follow its parents back towards the root to find a scapegoat. The scapegoat is the first node (on the path from the newly inserted node towards the root) that satisfies

$$\frac{sizeOfSubtree(scapegoat.child)}{sizeOfSubtree(scapegoat)} > \frac{2}{3}$$

Here *scapegoat.child* is the child of the scapegoat on the path from the newly inserted node towards the root. Once you have found the scapegoat, rebuild the subtree rooted at the scapegoat into a balanced binary search tree. Be careful! You must balance the subtree, then graft the subtree's root into the place the scapegoat formerly occupied in the original tree. In particular, make sure you set the appropriate child reference in the scaoegoat's original parent to point to the now-balanced subtree's root.

2.2.8 `bool remove(const T &element);`

You should remove the node with the given `element` from this scapegoat tree and return `true`. Don't forget to make necessary pointer modifications in the tree. If there exists no such node in ~~the binary search tree~~ this scapegoat tree, this function should return `false`. ***There will be no duplicated elements in this binary search tree.***

After removing the node from the scapegoat tree with the help of ordering property of binary search trees, you should check the upper bound condition (1) of the scapegoat tree. At removal, you should not change the upper bound. If the upper bound condition is not violated, you should do nothing. If it is violated, you should rebuild the entire tree into a balanced binary search tree and reset upper bound to the number of nodes.

2.2.9 `void removeAllNodes();`

You should remove all nodes of this scapegoat tree so that the scapegoat tree becomes empty. Don't forget to make necessary pointer modifications in the tree.

2.2.10 `const T &get(const T &element) const;`

You should search this scapegoat tree for the node that has the same element with the given `element` and return ~~a pointer to that node~~ the element of that node. You can use the `operator==` to compare two `T` objects. If there exists no such node in this scapegoat tree, this function should throw `NoSuchItemException`. ***There will be no duplicated elements in this scapegoat tree.***

2.2.11 `void print(TraversalMethod tp=inorder) const;`

Given a traversal method (`inorder`, `preorder` or `postorder`) as parameter (namely `tp`), this function prints this scapegoat tree by traversing its nodes accordingly. The code for `inorder` printing is already given. You should complete this function for `preorder` and `postorder` printing. You should follow the printing format used for the given `inorder` printing to implement the others.

2.2.12 `ScapegoatTree<T> &operator=(const ScapegoatTree<T> &rhs);`

This is the overloaded assignment operator. You should remove all nodes of this scapegoat tree and then, you should create new nodes by copying the nodes of the given `rhs` and insert those new nodes into this scapegoat tree. The structure among the nodes of the given `rhs` should also be copied to this scapegoat tree.

2.2.13 `void balance();`

This function manually balances this scapegoat tree. It rebuilds the entire tree into a balanced binary search tree by applying the following steps:

1. Output the elements of the tree into an array in sorted (ascending) order.
2. Build a new balanced binary search tree from the sorted array using the technique similar to a binary search:
   - Start from the entire array (*for example, [1, 3, 5, 7, 9]*),
   - Pick the middle element to make it a tree node (*5*),
   - Recurse on the left subarray (*[1, 3]*),
   - Recurse on the right subarray (*[7, 9]*).

2.2.14 `const T &getCeiling(const T &element) const;`

You should search this scapegoat tree for the node associated with the least element greater than or equal to the given `element` and return the element of that node. You can use the `operator==` to compare two `T` objects. If there exists no such node in this scapegoat tree (i.e., all nodes have elements less than the given `element`), this function should throw `NoSuchItemException`. ***There will be no duplicated elements in this scapegoat tree.***

2.2.15 `const T &getFloor(const T &element) const;`

You should search this scapegoat tree for the node associated with the greatest element less than or equal to the given `element` and return the element of that node. You can use the `operator==` to compare two `T` objects. If there exists no such node in this scapegoat tree (i.e., all nodes have elements greater than the given `element`), this function should throw `NoSuchItemException`. ***There will be no duplicated elements in this scapegoat tree.***

2.2.16 `const T &getMin() const;`

You should search this scapegoat tree for the node associated with the least element and return the element of that node. You can use the `operator==` to compare two `T` objects. If there exists no such node in this scapegoat tree (i.e., the scapegoat tree is empty), this function should throw `NoSuchItemException`. ***There will be no duplicated elements in this scapegoat tree.***

2.2.17 `const T &getMax() const;`

You should search this scapegoat tree for the node associated with the greatest element and return the element of that node. You can use the `operator==` to compare two `T` objects. If there exists no such node in this scapegoat tree (i.e., the scapegoat tree is empty), this function should throw `NoSuchItemException`. ***There will be no duplicated elements in this scapegoat tree.***

2.2.18 `const T &getNext(const T &element) const;`

You should search this scapegoat tree for the node associated with the least element greater than the given `element` and return the element of that node. You can use the `operator==` to compare two `T` objects. If there exists no such node in this scapegoat tree (i.e., all nodes have elements less than or equal to the given `element`), this function should throw `NoSuchItemException`. ***There will be no duplicated elements in this scapegoat tree.***

3 Tree Map Implementation (30 pts)

Tree map is a scapegoat tree based map implementation that keeps its entries sorted according to the natural ordering of ~~its~~ their keys. Tree maps are used to store data ~~values~~ in key:value pairs. A tree map is a collection which is ordered (i.e., the entries have a defined order and that order will not change), changeable (i.e., we can update, add or remove entries after the tree map has been created) and ***do not allow duplicates (i.e., tree maps cannot have two ~~items~~ entries with the same key)***. Tree map entries are presented in key:value pairs, and can be referred to by using the ~~key name~~ keys.

The tree map in this assignment is implemented as the class template `TreeMap` with the template argument `K`, which is used as the type of keys stored in the map, and `V`, which is used as the type of values stored in the map. `TreeMap` class has a `ScapegoatTree` object in its private data field (namely `stree`) with the type `KeyValuePair<K, V>`. This `ScapegoatTree` object keeps the key-value mappings (or pairs) in the tree map. `KeyValuePair` class represents the key-value mappings in the tree map.

The `TreeMap` and `KeyValuePair` classes has their definitions in *TreeMap.h* and *KeyValuePair.h* files, respectively.

3.1 `KeyValuePair` Class

`KeyValuePair` class represents key-value mappings that constitute tree maps. A `KeyValuePair` object keeps a variable of type `K` (namely `key`) to hold the key, and a variable of type `V` (namely `value`) to hold the value. The class has three constructors, overloaded relational operators, getters, setters, and the overloaded output operator. They are already implemented for you. Note that the overloaded relational operators compare only keys of the objects to decide. You should not change anything in file *KeyValuePair.h*.

3.2 `TreeMap` Class

In `TreeMap` class, all member functions should utilize `stree` member variable to operate as described in the following subsections. In ~~*TreeMap.cpp*~~ *TreeMap.h* file, you need to provide implementations for following functions declared under *TreeMap.h* header to complete the assignment.

3.2.1 `void clear();`

This function removes all key-value mappings from this map so that it becomes empty.

3.2.2 `const V &get(const K &key) const;`

This function returns the value of the key-value mapping specified with the given `key` from this map. If there exists no such key-value mapping in this tree map, this function should throw `NoSuchItemException`.

3.2.3 `bool pop(const K &key);`

This function removes the key-value mapping specified with the given `key` from this map and returns `true`. If there exists no such key-value mapping in this tree map, this function should return `false`.

3.2.4 `bool update(const K &key, const V &value);`

This function adds a new key-value mapping specified with the given `key` and `value` to this map and returns `true`. It takes the mapping information (`key` and `value`) as parameter and inserts a new `KeyValuePair` object to the `stree` scapegoat tree. If there already exists a mapping with the given `key` in this tree map, this function should return `false`.

3.2.5 `const KeyValuePair<K, V> &ceilingEntry(const K &key);`

This function returns a key-value mapping from this tree map associated with the least key greater than or equal to the given `key`. If there exists no such key-value mapping ~~with the given key~~ in this tree map, this function should throw `NoSuchItemException`. ~~***There will be no key-value pairs with same key in the tree map.***~~

3.2.6 `const KeyValuePair<K, V> &floorEntry(const K &key);`

This function returns a key-value mapping from this tree map associated with the greatest key less than or equal to the given `key`. If there exists no such key-value mapping ~~with the given key~~ in this tree map, this function should throw `NoSuchItemException`. ~~***There will be no key-value pairs with same key in the tree map.***~~

3.2.7 `const KeyValuePair<K, V> &firstEntry();`

This function returns a key-value mapping from this tree map associated with the least key. If there exists no such key-value mapping in this tree map, this function should throw `NoSuchItemException`.

3.2.8 `const KeyValuePair<K, V> &lastEntry();`

This function returns a key-value mapping from this tree map associated with the greatest key. If there exists no such key-value mapping in this tree map, this function should throw `NoSuchItemException`.

3.2.9 `void pollFirstEntry();`

This function removes the key-value mapping associated with the least key from this map. If there exists no such key-value mapping in this tree map, this function should throw `NoSuchItemException`.

3.2.10 `void pollLastEntry();`

This function removes the key-value mapping associated with the greatest key from this map. If there exists no such key-value mapping in this tree map, this function should throw `NoSuchItemException`.

3.2.11 `std::vector<KeyValuePair<K, V> > subMap(K fromKey, K toKey) const;`

This function returns a portion of this map whose keys range from `fromKey`, inclusive, to `toKey`, inclusive as `std::vector` of key-value mappings. For this function, you may assume that there are key-value mappings in this map with keys `fromKey` and `toKey`, and you may also assume that `fromKey` is less than or equal to `toKey`.

4 Driver Programs

To enable you to test your `ScapegoatTree` and `TreeMap` implementations, two driver programs, *main_scapegoatt* and *main_treemap.cpp* are provided.

5 Regulations

1. **Programming Language:** You will use C++.
2. **S**tandard Template Library is **not** allowed unless you are explicitly asked to use it for a task.
3. **E**xternal libraries other than those already included are **not** allowed.
4. **T**hose who do the operations (insert, remove, get, ...) without utilizing the binary search tree will receive **0 grade**.
5. **T**hose who modify already implemented functions and those who insert other data variables or public functions and those who change the prototype of given functions will receive **0 grade**.
6. **Y**ou can add private member functions whenever it is explicitly allowed.
7. **Late Submission Policy:** Each student receives 5 late days for the entire semester. You may use late days on programming assignments, and each allows you to submit up to 24 hours late without penalty. For example, if an assignment is due on Thursday at 11:30pm, you could use 2 late days to submit on Saturday by 11:30pm with no penalty. Once a student has used up all their late days, each successive day that an assignment is late will result in a loss of 5% on that assignment.

   No assignment may be submitted more than 3 days (72 hours) late without permission from the course instructor. In other words, this means there is a practical upper limit of 3 late days usable per assignment. If unusual circumstances truly beyond your control prevent you from submitting an assignment, you should discuss this with the course staff as soon as possible. If you contact us well in advance of the deadline, we may be able to show more flexibility in some cases.
8. **Cheating:** We have zero tolerance policy for cheating. In case of cheating, all parts involved (source(s) and receiver(s)) get zero. People involved in cheating will be punished according to the university regulations. Remember that students of this course are bounded to code of honor and its violation is subject to severe punishment.
9. **Newsgroup:** You must follow the Forum (`odtuclass.metu.edu.tr`) for discussions and possible updates on a daily basis.

6 Submission

- Submission will be done via CengClass (`cengclass.ceng.metu.edu.tr`).
- Don't write a main function in any of your source files.
- A test environment will be ready in CengClass.
  - You can submit your source files to CengClass and test your work with a subset of evaluation inputs and outputs.
  - Additional test cases will be used for evaluation of your final grade. So, your actual grades may be different than the ones you get in CengClass.
  - Only the last submission before the deadline will be graded.