Middle East Technical University Department of Computer Engineering

CENG 213

Data Structures

Fall 2017-2018

Programming Assignment 2

Due date: 10 December 2017, Sunday, 23:55

1 Objectives

In this assignment you are expected to implement a generic AVL tree of key-value pairs and specialized function objects to compare keys, also including a tree iterator class offering inorder traversal and use this data structure to build a bookstore as a collection of books that can be indexed with respect to distinct keys and comparison schemes.

Keywords: *Binary Search Tree, AVL Tree, Tree Iterators*

2 Binary Search Tree Class Template (40 pts)

Outline of `BinarySearchTree` class template implemented in *bst.hpp* file is summarized in the following.

```
template <typename Key, typename Object,
          typename Comparator=std::less<Key> >
class BinarySearchTree
{
  private:
    struct BinaryNode
    { /*...*/ };
  public:
    class Iterator
    { //...
      friend class BinarySearchTree<Key, Object, Comparator>;
    };
  public:
    //public BinarySearchTree methods
  private: //data
    BinaryNode * root;
    size_t nodes;
    Comparator isLessThan;
  private:
    /* private utility functions */
};
```

The AVL tree used in this assignment is implemented as the class template `BinarySearchTree` with template arguments `Key`, `Object`, and `Comparator` having the default value of `std::less<Key>` defined under `<functional>`. These arguments will be discussed in detail throughout the assignment text.

The basic building block of our AVL tree is the binary node structure implemented as `BinaryNode` nested `struct` type placed under `private` section, resulting in making all of its data and functions being publicly available to `BinarySearchTree`, yet totally hiding its existence from the outside world. Next, an `Iterator` class to materialize the notion of position of a node inside the tree is nested under `public` section declaring `BinarySearchTree` to be a friend so that the tree may access all of its sections freely, and the outside world may access its `public` functionality by declaring variables of its type as `BinarySearchTree<K,O,C>::Iterator`, where `K`, `O`, and `C` are concrete types instantiating the AVL template.

The interface of `BinarySearchTree` class comprises of functions included under interleaving `public` sections, whose implementations heavily rely on `private` utility functions as we will inspect shortly. Also, the data members of the tree is placed under the `private` section where we have the designated **root pointer** of the tree through which rest of the nodes can be accessed, a variable to store the number of nodes in the tree, and a `Comparator` type function object to compare different keys.

2.1 `BinaryNode`

This type represents a node in the AVL tree and consists of a `Key` type variable **uniquely** identifying the node, an `Object` type data to be stored within the node, two pointers to left and right subtrees, and an additional attribute to record the height of the node; all of which can be initialized by the declared constructor which has been implemented in lines following the type declaration of `BinarySearchTree`. Do not change its implementation in parts of the *bst.hpp* file.

2.2 `Iterator`

The AVL `Iterator` embodies a pointer to some `BinaryNode` through which `Object` data inside the node can be accessed. It encapsulates a pointer to a node and a pointer to the root of the tree to which it belongs as its data members `current` and `root` respectively. Moreover, it contains `s`, a stack of pointers to nodes, which should be used to carry out inorder traversal of the tree on condition that `useStack` variable is set to `true`. Default constructor and comparision operators have been implemented. Going over indicated parts in *bst.hpp*, you are expected to complete the remaining functionality in accordance with the following specifications.

2.2.1 `const Object & operator*( ) const;`

`Object` type object stored in the node adressed by `current` pointer must be returned for view-only purposes.

2.2.2 `Object & operator*( );`

`Object` type object stored in the node addressed by `current` pointer must be returned so that its value may be modified.

2.2.3 `Iterator(BinaryNode *, const BinarySearchTree &, bool);`

This private constructor will only be invoked by the member functions of the `BinarySearchTree` class declared to be a friend and besides **initializing** data members of the `Iterator` object with corresponding constructor arguments, if `useStack` variable is set as `true` (note that its default value is `true`), it will update `current` pointer variable to store the address of the first visited `BinaryNode` when inorder traversal is applied to the subtree that the original `BinaryNode *` type argument of the constructor roots (when dereferenced). Regarding the condition in which `useStack` is `true`, it is advisable to complete the implementation of the constructor after you implement `operator++` described subsequently.

2.2.4 `Iterator & operator++( );`

If `useStack` variable has been set the value `false`, then `current` pointer must be updated as `NULL`. Otherwise, `current` pointer must be updated to store the address of the `BinaryNode` that is the **inorder successor** of then-current node. If there exists no inorder successor of the current node, then `current` must be set as `NULL`.

In all cases, a self reference should be returned. Regarding inorder traversal, it is highly advisable that you use the C++ STL `std::stack<BinaryNode *>` type member variable `s`. A practical use of `Iterator` class for inorder traversal can be seen within `traverseDataItems` function implemented in *test_tree.cpp* driver program.

2.3 BinarySearchTree

You are going to use `BinarySearchTree` class template to insantiate types of objects that provide you with the required AVL tree functionality throughout this assignment. Previously, data members have been briefly described. Their use will be elaborated in the context of utility functions discussed in the following.

Constructor, destructor, `find`, `begin`, `end`, `height`, `size`, `empty` and `print` functions that reside within the `public` section of the class have already been **implemented** vastly in accordance with your textbook. You must figure out what they are computing by inspecting the code and you must **not** modify these implementations. You may want to pay specific attention to the implementation of the destructor, `find`, `height` and `print` functions that utilize **recursive** `private` utility functions that usually begin their computation at the designated `root` position. This programming style may be helpful in your coding of the uncompleted functions as you **can** add other functions to the `private` part of the class. Alternatively, you may devise iterative solutions.

Copy constructor and assignment operator signatures are also included in the `private` section. We will not allow their use in this assignment and in order to block compiler defaults, prototypes are provided and you should perform **no** implementation. This is a trick utilized in C++ world so as to make objects of the class type uncopiable or unassignable.

AVL tree rotation functions `rotateWithLeftChild`, `rotateWithRightChild`, `doubleWithLeftChild`, and `doubleWithRightChild` have been implemented under `private` section. Use them when you need to rebalance your tree in insertion and removal procedures.

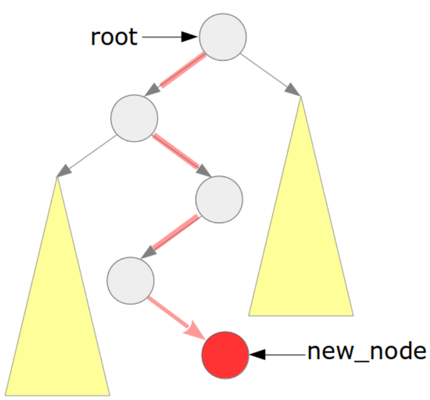
You need to use `Comparator` type data member `isLessThan` to compare two `Key` variables `k1` and `k2` by issuing a call as `isLessThan(k1, k2)` that returns a bool variable. These type of classes with overloaded `operator()` are called *function objects* that offer an alternative to the use of C-style function pointers that empower us with custom comparision functions in building our `BinarySearchTree`. The default value of `Comparator` template argument is `std::less<Key>` which internally translates calls of the format `isLessThan(k1, k2)` into `operator<(k1, k2)`. For more details, read Section 1.6.4 of your textbook.

You must provide implementations for the following `public` interface methods that have been declared under indicated portions of *bst.hpp* file. For examples of the use of these functions inspect the expected output of the program in *test_tree.cpp* file.

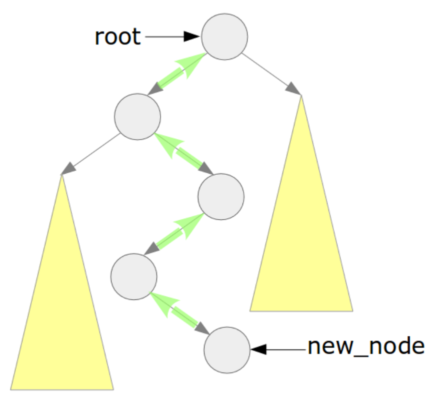
2.3.1 `void insert(const Key &, const Object &);`

Conduct the recursive search pattern adapted by `find` function to locate the correct place to insert the `Key-Object` pair arguments. If the `Key` does exist in the tree, **update** the data element stored in the node in which the `Key` value resides with the value of the `Object` parameter.

If the `Key` does not exist, create a `new_node` with the `Key-Object` pair and increment the number of nodes. As shown in Figure 1a, the path traveled to find the correct place to insert the `new_node` into the tree is marked with red lines and at the end you **must** set up the pointer connection shown by the red arrow so that `new_node` can be accessed via a unique path beginning at the `root`.



(a) First phase of insertion



(b) Prior to rebalancing

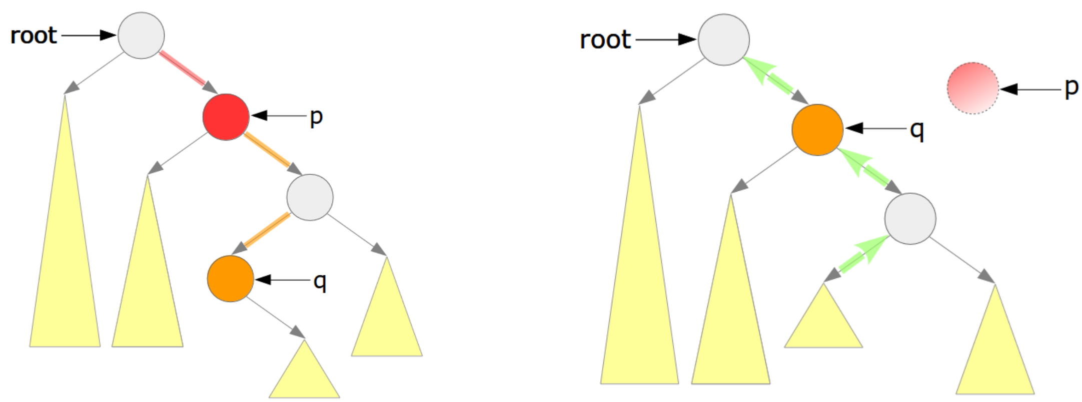
Figure 1: Insertion example when the parameter key is not present within the tree

You **must** update heights of the nodes that reside in the path that `insert` function travels beginning at the designated `root` position up to the newly inserted node in **reverse** order and **rebalance** all these mentioned nodes in case the absolute value of difference between heights of the left and right subtrees they root **exceeds one**, in accordance with the AVL tree specifications. Having inserted the `new_node`, the nodes on the path marked by green arrows in Figure 1b **must** be subject to these height updates and rebalancing.

2.3.2 `void remove(const Key &);`

You must find the node holding the `Key` parameter by exploiting binary search tree characteristics via a recursive search starting from the `root` position. If the `Key` parameter does not exist, do **not** do anything. Otherwise if the node holding the `Key` parameter is a leaf, then directly delete this node. If the node has one child, set up pointer connections with the parent and the child of the node and delete the node. Whenever you delete a node, decrement the number of nodes.

The case in which the node to remove has two children is more complicated. As shown in Figure 2a, following the path marked with orange arrows, identify the node `q` which is the inorder successor of the node to remove `p` and **place** `q` into where `p` currently is. Do **not** copy `Key` and `Object` values between nodes as copying would invalidate all node iterators (i.e. pointers to nodes) or copying might be costly and even might not be allowed. Instead, exchange pointers to nodes and update interconnecting pointers by carefully analyzing possible cases.



(a) First phase of removal

(b) Prior to rebalancing

Figure 2: Removal of a node having two children

At this point, you can delete the node pointed by `p` which is no longer accessible via the `root` as shown in Figure 2b and decrement the number of nodes. You **must** make sure that only the node with the address `p` is deleted from the tree and the address of other nodes including `q` are kept **unchanged**.

You **must** update heights of the nodes that reside in the path that `remove` function has travelled via **backtracking** and **rebalance** all visited nodes in accordance with the AVL tree specifications. If the deleted node had two children, you should update heights and perform rebalancing of nodes that are on the path starting from the initial position of `q` up to the `root` marked by green arrows in Figure 2b. Note that there **might** be other cases than the one depicted in Figure 2 and your solution must be exhaustive, covering all of them.

2.3.3 `std::list<Iterator> find(const Key &, const Key &) const;`

You must return a C++ STL `list` of `Iterator` objects having their `useStack` variables set as `false` that correspond to positions of the nodes whose `Key` values fall into the range between the first and second parameters of this function. It is assumed that the first `Key` parameter `isLessThan` or equal to the second and that the interval is closed i.e. includes the two margins. Try to exploit binary search tree characteristics to **limit** the number of nodes you should visit in your inorder travel strategy to the order of the number of nodes within the interval for better efficiency.

3 Bookstore Implementation (60 pts)

3.1 `Book`

`Book` class represents the information needed to be stored for each item in the bookstore. A `Book` comprises of its `isbn`, `title`, `author`, `publisher`, publication `year`, number of `pages` and `status` attribute to indicate whether it is in stock or not. Particular accessors and mutators together with `operator<<` function for printing has been implemented.

Each `Book` can be **uniquely** identified by its `isbn` member. Also **no** two books with the same `name` and `title` may exist. These attributes can be set upon `Book` object construction and may

not be changed later. Note that `operator=` is specifically defined for `Book` due to `const` members.

Unique title and author pair is also regrouped into `Book::SecondaryKey` class type and variables of this type will be used in building secondary index trees. Inspect the complete implementations in *book.hpp* and *book.cpp* files. These files should **not** be changed.

3.2 `TitleComparator` and `AuthorComparator`

Implementation of `TitleComparator` type that intends to compare two `Book::SecondaryKey` objects primarily based on their titles must be completed in *title_comparator.hpp* file by coding only the inline `operator()(const Book::SecondaryKey &, const Book::SecondaryKey &) const` member function with respect to these specifications:

- Perform a **case-insensitive**, **lexicographic** comparision between the titles of two key arguments.
- If the first title comes before the second return `true`.
- If the two titles are equal, then compare the two authors and return `true` only if the first author comes before the second.
- Return `false` otherwise.

You must finish definition of another class called `AuthorComparator` so that you may compare two `Book::SecondaryKey` objects primarily based on their authors. To do that, perform the same steps outlined in case of `TitleComparator` implementation with one imperative modification: comparing authors first instead of the titles in *author_comparator.hpp* file. You may test these comparators through examples included in *main_book.cpp* file.

3.3 `BookStore` Data

`BookStore` objects internally keep three `BinarySearchTree` indices with identifiers `primaryIndex`, `secondaryIndex` and `ternaryIndex`. Note that their types are aliased with shorter names using `typedef` constructs under the `private` section.

Notice that `primaryIndex` of shorter type name `BSTP` will store actual `Book` objects and will use the default lexicographic ordering of `isbn` values to build itself and yet `secondaryIndex` of new type name `BSTS` and `ternaryIndex` of new type name `BSTT` will both rely on secondary key objects of new type name `SKey` values, and these values will be ordered utilizing `AuthorComparator` and `TitleComparator` classes respectively. Pay specific attention to the declaration that states `BSTS` and `BSTT` trees take pointers to (`const`) `Book` data stored in `BSTP`.

3.4 `BookStore` Interface

Default constructor and `printPrimarySorted`, `printSecondarySorted` and `printTernarySorted` functions to print books under different indices using inorder binary search tree iterators have been already implemented. Do **not** change these functions.

In *bookstore.cpp* file, you need to provide implementations for following functions declared under *bookstore.hpp* header to complete the assignment.

3.4.1 `void insert(const Book &);`

Insert the parameter `Book` object into **all** three AVL tree indices. Note that actual data will be stored inside some node of `primaryIndex` as a copy of the parameter of this function and the other indices will receive the address of the location corresponding to data portion of the `primaryIndex` node and store this as a pointer variable within their nodes. Consequently indices based on `SKey` will hold up less space in total.

3.4.2 `void remove(const std::string &);`

Remove the book corresponding to the parameter `isbn` value from **all** indices if applicable.

3.4.3 `void remove(const std::string &, const std::string &);`

Remove the book corresponding to the parameter `title` and `author` values in order from **all** indices if applicable.

3.4.4 `void removeAllBooksWithTitle(const std::string &);`

Remove **all** books with the parameter `title` from **all** indices when applicable. You can safely assume that author names reside within the closed range of `"a"` and `"{"`.

3.4.5 `void makeAvailable(const std::string &);`

Update the `status` variable of the `Book` object with the parameter `isbn` value as `true` if applicable so as to indicate that the book is now available in the store.

3.4.6 `void makeUnavailable(const std::string &, const std::string &);`

Update the `status` variable of the `Book` object with the given `title` and `author` values in order as `false` if applicable to signal that the book is no longer available in the store.

3.4.7 `void updatePublisher(const std::string &, const std::string &);`

Update `publisher` fields of all `Book` objects whose `author` field value is equal to the first parameter as the second parameter of this function. You can safely assume that book titles reside within the closed interval of `"a"` and `"{"`. This function will be invoked whenever the author makes a deal with some other publisher to have all their books reprinted.

3.4.8 `void printBooksWithISBN(const std::string &, const std::string &, unsigned short) const;`

Print each `Book` object whose `isbn` member falls within the closed interval of the first and the second parameter of this function onto the console followed by an `std::endl` IO manipulator if they were published **no** earlier than the third parameter of the function whose default value is 0.

3.4.9 `void printBooksOfAuthor(const std::string &, const std::string &, const std::string &);`

Print each `Book` object whose `author` member is equal to the first parameter and `title` member falls within the closed range of the second and the third parameters of the function onto the console followed by `std::endl`. Note that the second and the third parameters receive default values of `"a"` and `"{"`, i.e. all books of the author are printed in this case.

4 Driver programs

For testing `BinarySearchTree` functionality, a driver program under the name *test_tree.cpp* has been provided. To see `TitleComparator` and `AuthorComparator` objects in action, *main_book.cpp* may be compiled and run. Tests regarding the *Bookstore* class, *test_bookstore.cpp* may be used. Expected outputs of driver programs are also given in separate **.out* files.

5 Regulations

1. **Programming Language:** You will use C++.
2. **S**tandard Template Library is allowed only for `list` and `stack`.
3. **E**xternal libraries other than those already included are not allowed.
4. **T**hose who do search, update, remove operations without utilizing the 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. **T**hose who use STL `vector` or compile-time arrays or variable-size arrays (not existing in ANSI C++) will receive 0 grade. Options used for `g++` are `-ansi -Wall -pedantic-errors -O0`.
7. **Y**ou can add private member functions whenever it is explicitly allowed.
8. **Late Submission:** You have totally 7 days of late submission for all programming assignments, and the latest date you may submit this assignment is 3 days after the deadline.
9. **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.
10. **R**emember that students of this course are bounded to code of honor and its violation is subject to severe punishment.
11. **Newsgroup:** You must follow the newsgroup (news.ceng.metu.edu.tr) for discussions and possible updates on a daily basis.

6 Submission

- Submission will be done via Moodle.
- Do not write a *main* function in any of your source files.
- A test environment will be ready in Moodle.
  - You can submit your source files to Moodle and test your work with a subset of evaluation inputs and outputs.
  - Additional test cases will be used for evaluation of your final grade, and only the last submission before the deadline will be graded.