Middle East Technical University | Department of Computer Engineering

CENG 213

Data Structures

Fall 2020-2021

Programming Assignment 2

Version 2 (December 18)

Due date: 24 December 2020, 23:55

1 Objectives

In this assignment you are expected to implement a generic binary search tree of key-value pairs and specialized function objects to compare keys, which is to be balanced by reorganizing itself as a complete tree with minimum height under distinct conditions indicated by associated function objects; and use this data structure to build a movie store as a collection of movies that can be indexed with respect to distinct keys and comparison schemes.

Keywords: *Binary Search Tree, Complete Tree, Generic Balancing*

2 Binary Search Tree Class Template (50 pts)

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

```
template <typename Key, typename Object,
          typename BalanceCondition=DefaultBalanceCondition,
          typename Comparator=std::less<Key> >
class BinarySearchTree
{
  public:
    struct Node
    {
       /* */
    };
  public:
    //public methods of BinarySearchTree
  private:
    Node * root;
    size_t numNodes;
    Comparator isLessThan;
    BalanceCondition isBalanced;
  private:
    //utility functions
};
```

The generic binary search tree (BST) used in this assignment is implemented as the class template `BinarySearchTree` with template arguments `Key`, `Object`, `BalanceCondition`, and `Comparator`, with the last two having the default values of `DefaultBalanceCondition`, and `std::less<Key>` defined in `<functional>`, respectively. These arguments will be discussed in detail throughout the assignment text.

The basic building block of our BST is the binary node structure implemented as `Node` nested `struct` type placed under `public` section, resulting in making all of its data and functions being publicly accessible as `BinarySearchTree<K,O,B,C>::Node`, where `K`, `O`, `B`, and `C` are concrete types instantiating the BST 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, a `Comparator` type function object to compare different keys, and a `BalanceCondition` type object to indicate whether the BST requires height-balancing following changes to its content through operations such as `insert` and `remove`.

2.1 `Node`

This type represents a node in the BST 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, an additional `height` attribute to record the height of the node, and finally a variable with identifier `subsize` that holds the number of the nodes the node roots including itself; 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 BinarySearchTree

You are going to use `BinarySearchTree` class template to instantiate types of objects that provide you with the required search 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.

Zero-parameter constructor, destructor, `getRoot`, `find`, `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 variables and 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**.

You need to use `Comparator` type data member `isLessThan` to compare two `Key` objects `k1` and `k2` by issuing a call as `isLessThan(k1, k2)` that returns a bool variable. These type of objects are called *function objects* that offer an alternative to the use of C-style function pointers that empower us with custom comparison 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.

Similar to the use of `Comparator`, another function object type called `BalanceCondition` is used to compare a BST's actual height with its ideal height, i.e. the minimum height it can attain, via overloaded `operator()` to indicate whether the BST is considered as balanced or not. A class called `DefaultBalanceCondition` is implemented just above the `BinarySearchTree`, which provides the default version of the template `BalanceCondition` as the name suggests. Instances of this class simply returns `true`, in accordance with the standard BST, in which balance is disregarded. Do not change the implementation of this class. You must implement the following constructor and `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 *main_tree.cpp* file.

2.2.1 `BinarySearchTree(const std::list<std::pair<Key, Object> > &);`

Given that the key object pairs in the list argument are sorted in ascending order with respect to `isLessThan` function object provided as a class template argument whose default is the ordinary `operator<` on `Key` values; this one-argument constructor builds up a **complete** tree comprising of number of nodes equal to the size of the list, and populates its element values in accordance with the inorder traversal of the tree and the order of items in the list. In other words, here you are expected to create a minimum-height binary search tree that is complete with its nodes getting its constructing key-data values from corresponding sorted items in the list. In addition, determine the height of each node as well as the number of nodes they root. For every test case, list object will be sorted. Do **not** attempt sorting it. Also note that complete tree will generate nodes in each level using an order from **left to right**. No other orders or BST's that are isomorphic to the minimum-height complete BST will be accepted so as to avoid ambiguities and ensure determinism. Expected time complexity of this operation is linear in tree size.

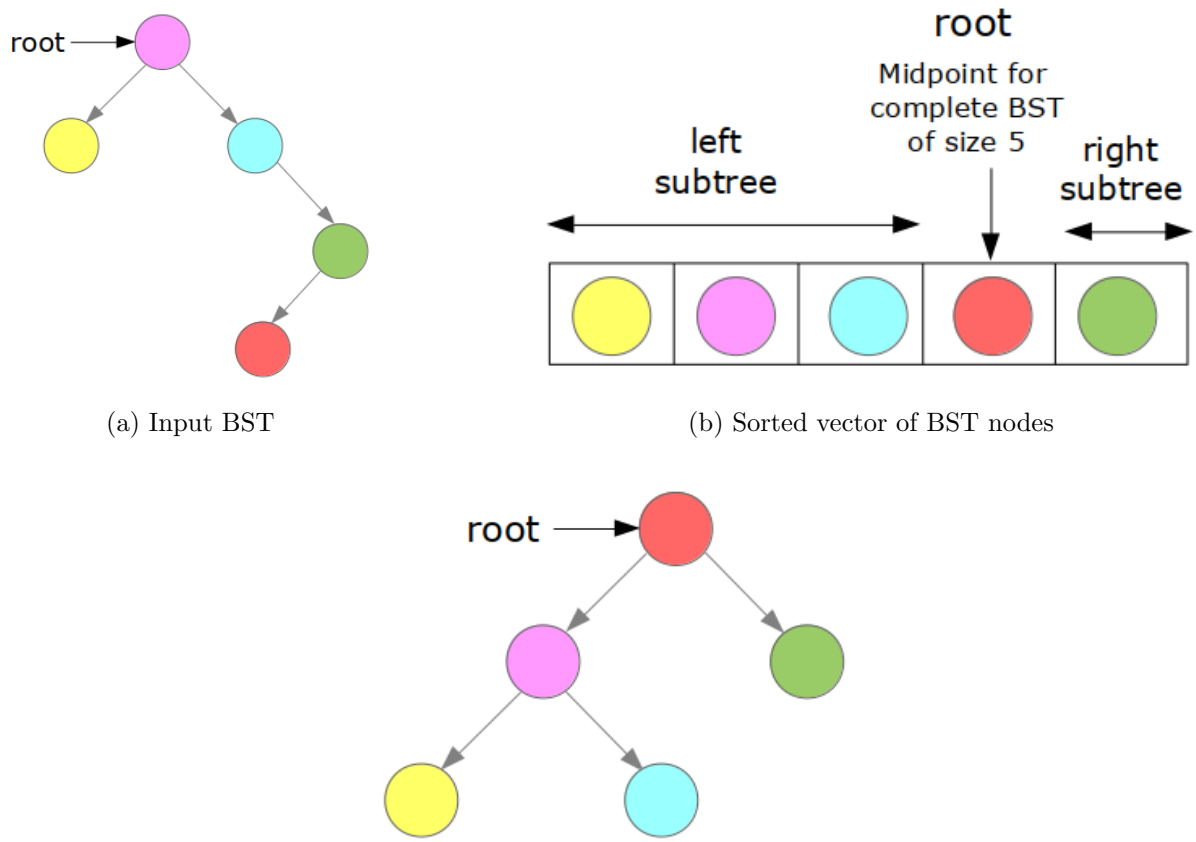
2.2.2 `void toCompleteBST();`

Current BST should be converted into a **minimum-height complete BST**, populated from **left to right** in each level, and keeping the memory adresses of the nodes of the input tree **unchanged**. Consequently, you should **not** create an auxiliary BST and populate its nodes by values from the original one, since copying of data can be disallowed depending on type properties. Expected time complexity of this operation is linear in tree size.

You can devise a recursive approach to carry out the conversion. An illustrated example is included in Figure 1. A non-complete BST of size 5 in Figure 1a is converted into a complete BST of size 5 in Figure 1c, without changing node locations indicated by distinct colors. In order to make this happen, you need an array of node pointers sorted with respect to the ordering in the input BST, as depicted in Figure 1b. Then, you must determine the index of the root for a complete BST of respected size. Here, you must pay attention to simply not selecting the mean or median index, since complete BST's are populated from left to right in each level, consequently exhibiting skewed distributions. Having correctly determined the root position, you must recursively carry out the outlined procedure for nodes with indices lower than the root index that constitute the left subtree, and for nodes with indices higher than the root index that build up the right subtree un-

til base cases are encountered. At the end, you should have the complete tree structure in Figure 1c.

You can implement other algorithms that do not violate the outlined restrictions.



(a) Input BST

(b) Sorted vector of BST nodes

(c) Output complete BST

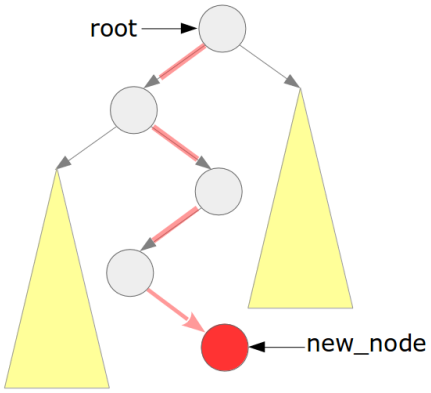
Figure 1: Conversion of a BST of size 5 into a complete BST

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

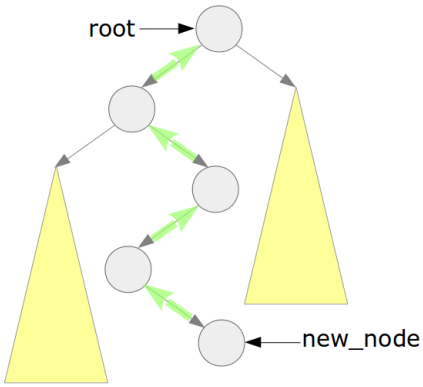
Conduct the recursive search pattern adapted by `find` function to locate the correct place to insert the `Key-Object` pair arguments. If the `Key` does exist in the tree, **update** the data element stored in the node in which the `Key` value resides with the value of the `Object` parameter.

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 2a, in which 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`.

You **must** update heights and subsizes 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 `BalanceCondition` type `isBalanced` variable's `operator()` returns `false` when provided with then-current height and the ideal height of the subtree in question. For nodes that root less than or equal to 1 nodes, no balancing is required.



(a) First phase of insertion



(b) Prior to rebalancing

Figure 2: Insertion example when the parameter key is not present within the tree

If a node roots N nodes including itself as a subtree T,

$$H_{ideal}(T) = \log_2(N), \quad N > 1 \tag{1}$$

the minimum height T can have is calculated using (1), and is referred to as the ideal height.

Then, invoking `isBalanced(`T`.root->height,`$H_{ideal}(T)$`)` yields the answer to the question if the BST is balanced at specified position. If not, T must be converted into a complete BST as indicated previously, and heights and subsizes of nodes in T must be updated.

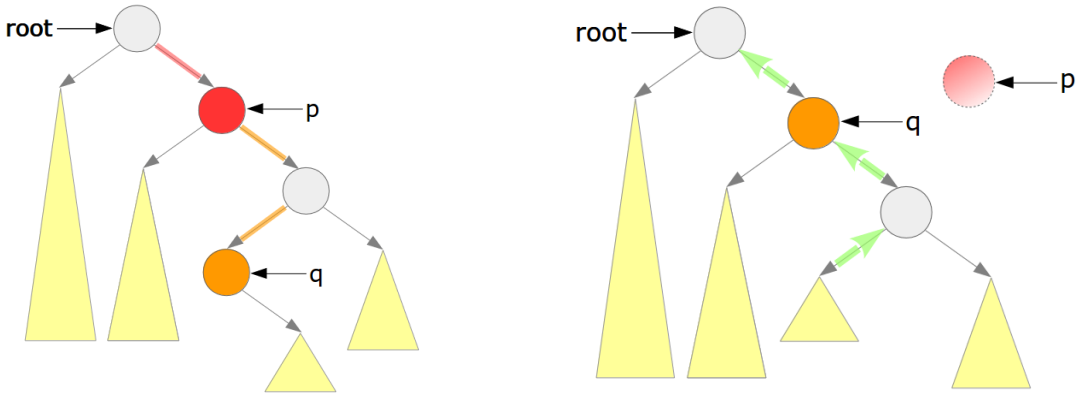
Having inserted the `new_node`, the nodes in the path marked by green arrows in Figure 2b **must** be subject to these height and subsize updates and rebalancing. If no balancing is required, expected time complexity of this operation is logarithmic in tree size on condition that a reasonable balance condition is adapted. The worst case of balancing tends to exhibit linear time complexity, however in test cases for bigger trees this will not occur frequently.

2.2.4 `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 3a, 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.

At this point, you can delete the node pointed by `p` which is no longer accessible via the `root` as shown in Figure 3b 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**.



(a) First phase of removal (b) Prior to rebalancing

Figure 3: Removal of a node having two children

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 BST specifications outlined so far. 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 3b. Note that there **are** cases other than the one depicted in Figure 3 and your solution must be exhaustive, covering all of them. If no balancing is required, expected time complexity of this operation is logarithmic in tree size on condition that a reasonable balance condition is adapted. The worst case of balancing tends to have linear time complexity, however in test cases for bigger trees this will not occur frequently.

2.2.5 `std::list<Node *> find(const Key &, const Key &) const;`

You must return a C++ STL `list` of `Node *` variables that store addresses 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. Consequently, if you are searching a small interval given the whole range of keys in the BST, you need to limit your access to more or less only the node nodes within range. Unrelated keys should not be visited exhaustively.

3 Movie Store Implementation (50 pts)

You need to have the `BinarySearchTree` implementation successfully complete to get full credits from this part, since the following `MovieStore` constitutes an application relying on the BST as the most crucial data structure employed within.

3.1 `Movie`

`Movie` class represents the information needed to be stored for each item in the movie store. A `Movie` comprises of its `id`, `title`, `director`, `company`, production `year`, its `duration` in minutes (rounded) 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 `Movie` can be **uniquely** identified by its `id` member. Also **no two** movies with the same `title` and `directory` may exist. These attributes can be set upon `Movie` object construction and may **not** be changed later. Note that `operator=` is specifically defined for `Movie` due to `const` members.

Unique title and director pair is also regrouped into `Movie::SecondaryKey` class type and variables of this type will be used in building secondary index trees. Inspect the complete implementations in *movie.hpp* and *movie.cpp* files. These files should **not** be changed. On Moodle, they are provided merely for view-only purposes with suffixes *-copy* in their respective file names, and any changes to these files will be disregarded during evaluation.

3.2 `TitleComparator` and `DirectorComparator`

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

- Perform a **case-insensitive**, **lexicographic** comparison 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 directors and return `true` only if the first director comes before the second.
- Return `false` otherwise.

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

3.3 `MovieStore` Data

`MovieStore` 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 `MSTP` will store actual `Movie` objects and will use the default lexicographic ordering of `id` values to build itself and yet `secondaryIndex` of new type name `MSTS` and `ternaryIndex` of new type name `MSTT` will both rely on secondary key objects of new type name `SKey` values, and these values will be ordered utilizing `DirectorComparator` and `TitleComparator` classes, respectively. Pay specific attention to the declaration that states `MSTS` and `MSTT` trees take pointers to (`const`) `Movie` data stored in `MSTP`. Balance condition for all these trees are implemented in `MovieBalanceCondition` class in *moviestore.h* file.

3.4 `MovieStore` Interface

Default constructor and `printPrimarySorted`, `printSecondarySorted` and `printTernarySorted` functions to print movies under different indices using inorder binary search tree iterators have been

already implemented. Do **not** change these functions.

In *moviestore.cpp* file, you need to provide implementations for following functions declared under *moviestore.hpp* header to complete the assignment. The header itself should not be changed, and any changes you perform on it will be disregarded during evaluation.

3.4.1 `void insert(const Movie &);`

Insert the parameter `Movie` object into **all** three BST 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 that 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 movie corresponding to the parameter `id` value from **all** indices if applicable.

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

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

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

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

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

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

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

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

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

Update `company` fields of all `Movie` objects whose `director` field value is equal to the first parameter as the second parameter of this function. You can safely assume that movie titles reside within the closed interval of `"a"` and `"z"`. This function will be invoked whenever the director makes a deal with some other company to have all their movies reproduced.

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

Print each `Movie` object whose `id` 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 produced **no** earlier than the third parameter of the function whose default value is 0. Output should be `id`-sorted in increasing order.

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

Print each `Movie` object whose `director` 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 movies of the director are printed in this case. Output should be sorted in increasing order with respect to `director`-first comparison scheme described under `Movie` section.

4 Driver programs

For testing `BinarySearchTree` functionality, a driver program under the name *main_tree.cpp* has been provided. To see `TitleComparator` and `DirectorComparator` objects in action, *main_movie.cpp* may be compiled and run. Tests regarding the `MovieStore` class, *main_moviestore.cpp* may be used. For all the three driver programs, expected outputs are provided as text files with the same identifiers as their code files of *.out* extension.

5 Regulations

1. **Programming Language:** You will use C++.
2. **S**tandard Template Library (STL) is allowed only for `list`, `vector`, `stack`, and `queue` data structures. Parameters of functions expecting particular STL data structures cannot be changed.
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 compile-time arrays or variable-size arrays (not existing in ANSI C++) will receive 0 grade. Expected solution is coded using ANSI C++. However, you are allowed to use C++11 if you are familiar with the version. Options to be used for `g++` are `-std=c++11 -Wall -pedantic-errors -O0`. We will check memory leaks with `valgrind`.
7. **Y**ou can add private member functions whenever it is explicitly allowed.
8. **I**mplementation of `BinarySearchTree` class template constitutes a **prerequisite** for `MovieStore` implementation, and thus grading will be done accordingly.
9. **Late Submission:** Refer to the syllabus.
10. **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.
11. **R**emember that students of this course are bounded to code of honor and its violation is subject to severe punishment.
12. **Newsgroup:** You must follow announcements and forum on `odtuclass` for discussions and possible updates daily.

6 Submission

- Submission will be done via Moodle on `cengclass`.
- 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.