Task

Write a C/C++ application containing the following three classes and the *main()* function for testing and checking.

1. Class Date

Class *Date* is implemented by instructor. See files *Date.cpp* and *Date.h* from Instructor's stuff.

2. Class Item

```
class Item
{
private:
       char Group;          // Any from range 'A'...'Z'
       int Subgroup;        // Any from range 0...99
       string Name;         // Any, but not empty
       Date Timestamp;      // Any
       .............        // To do
public:
       Item();              // Fills the four fields above with preudo-random values
       Item(char, int, string, Date);
       Item(const Item &); // copy constructor
       ~Item();
       .............         // To do
};
```

Requirements:

1. To get random groups and subgroups use C++ *default_random_engine_generator*.
2. The random names must be taken from file *Birds.txt*[1] stored in Instructor's stuff. Read the file into memory, create a vector of strings and use C++ *default_random_engine_generator* to retrieve names.
3. To create a random date use method *Date::CreateRandomDate().* Tip: use intervals with width one year or even more.

3. Class Data

Class *Data* must contain a C++ STL container containing objects of class *Item*. Its methods must allow the user to see the contents of container and to edit it.

The private member implementing the container must be:

```
map<char, map<int, list<Item *> *> *> DataStructure;
```

It is a C++ map in which the *Item* members *Group* are the keys. The values are pointers to inner C++ maps in which the keys are *Item* members *Subgroup* and values are pointers to lists. The lists contain pointers to items.

Class *Data* must contain the following public methods:

[1] Downloaded from http://www.jimpalt.org/birdwatcher/name.html

1. `Data(int n);`
   Constructs the object and fills the container with *n* random items.
2. Data();
   Constructs the object with empty container;
3. `~Data();`
   Destructs the object and releases all the memory occupied by the container and the items in it.
4. `void PrintAll();`
   Prints all the items stored in the container in command prompt window in easily readable format (see Appendix). Items from the same group and subgroup must be ordered by their names.
5. `int CountItems();`
   Returns the total number of items in the container.
6. `map<int, list<Item *> *> *GetGroup(char c);`
   Returns the pointer to *map* containing all the items from group *c*. If the group does not exist, returns *nullptr*.
7. `void PrintGroup(char c);`
   Prints all the items from group *c* in command prompt window in easily readable format (see Appendix). Items from the same subgroup must be ordered by their names. If the group was not found, throws *invalid_argument_exception*.
8. `int CountGroupItems(char c);`
   Returns the current number of items in group *c*. If the group does not exist, returns 0.
9. `list<Item *> *GetSubgroup(char c, int i);`
   Returns the pointer to *list* containing all the items from subgroup *I* from group *c*. If the subgroup does not exist, returns *nullptr*.
10. `void PrintSubgroupByNames(char c, int i);`
    Prints all the items from subgroup i from group *c* in command prompt window in easily readable format (see Appendix). Items must be ordered by their names. If the subgroup was not found, throws *invalid_argument_exception*.
11. `void PrintSubgroupByDates(char c, int i);`
    Prints all the items from subgroup i from group *c* in command prompt window in easily readable format (see Appendix). Items must be ordered by their timestamps. If the subgroup was not found, throws *invalid_argument_exception*.
12. `int CountSubgroupItems(char c, int i);`
    Returns the current number of items in subgroup *i* from group *c*. If the subgroup does not exist, returns 0.
13. `Item *GetItem(char c, int i, string s);`
    Returns the pointer to the first of items specified by group *c*, subgroup *i* and name *s*. If the item was not found returns *nullptr*.
14. `void PrintItem(char c, int i, string s);`
    Prints the first of item specified by group *c*, subgroup *i* and name *s*. If the item was not found throws *invalid_argument_exception*.
15. `Item *InsertItem(char c, int i, string s, Date d);`
    Creates and inserts the specified item. Returns the pointer to new item. If the specified item already exists or the input parameters are not correct, returns *nullptr*. If necessary, creates the missing group and subgroup.

16. `list<Item *> *InsertSubgroup(char s, int i, initializer_list<Item *> items);`
    Creates and inserts the specified subgroup (i.e. list of pointers to items). The initializer_list contains pointers to new items. Returns the pointer to new list. If the specified subgroup already exists or the input parameters are not correct, returns *nullptr*. If necessary, creates the missing group.
17. `map<int, list<Item *> *> *InsertGroup(char c, initializer_list<int> subgroups, initializer_list<initializer_list<Item *>> items);`
    Creates and inserts the specified group (i.e. map in which the keys are *Item* members *Subgroup* and values are pointers to lists containing pointers to items). The *subgroups* initializer_list presents the keys to be included into the new map. The *items initializer_list* contains initalizer_lists presenting pointers to items to be included. The first initalizer_list from *items* corresponds to the first integer in *subgroups*. Returns the pointer to new map. If the specified group already exists or the input parameters are not correct, returns *nullptr*.
18. `bool RemoveItem(char c, int i, string s);`
    Removes the specified item. If after removing the subgroup has no members (i.e. its list is empty), remove it too. If after that the group (i.e. map) is empty, remove it also. All the not used memory must be released. Return value: *false* if the item was not found, otherwise *true*.
19. `bool RemoveSubgroup(char c, int i);`
    Removes the specified subgroup (i.e. list of pointers to items). If after removing the corresponding group (i.e. map) has no members, remove it too. All the not used memory must be released. Return value: *false* if the subgroup was not found, otherwise *true*.
20. `bool RemoveGroup(char c);`
    Removes the specified group. All the not used memory must be released. Return value: *false* if the group was not found, otherwise *true*.

General requirements

1. Start the project as Visual Studio Console Application.
2. Instead of simple *for* loops try to use range-based *for* loops, methods built into maps and lists and algorithms like *for_each* from *<algorithm>*.

Evaluation

To verify that all the methods of class *Data* are working correctly, the Visual Studio project must contain test functions implementing the test cases specified below.

The deadline is the lecture on week 10 (end of March). However, the students can present his / her results earlier.

If on week 10 the personal contacts between students and teachers are allowed, a student can get the assessment only if he / she attends personally. In that case electronically (e-mail, cloud, GetHub, etc.) sent courseworks are neither accepted nor reviewed. Presenting the final release is not necessary. It is OK to demonstrate the work of application in Visual Studio environment.

If on week 10 the personal contacts between students and teachers are not allowed, the students must upload his / her complete Visual Studio project into GitHub or or some other place in cloud and send the link.

Test cases

1. Create object containing 300 items. Use methods *PrintAll()* and *CountItems()* to check the created object.
2. Select a group. Condition: there must be at least one subgroup containing at least two items. Apply methods *PrintGroup()* and *CountGroupItems()* for the selected group.
3. From the selected group select a subgroup containing at least two items. Apply methods *PrintSubgroupByNames(), PrintSubgroupByDates()* and *CountSubgroupItems()* for the selected subgroup.
4. From the selected group select a subgroup containing only one item. Apply methods *PrintSubgroupByNames(), PrintSubgroupByDates()* and *CountSubgroupItems()* for the selected subgroup.
5. Apply methods *PrintSubgroupByNames(), PrintSubgroupByDates()* and *CountSubgroupItems()* for a non-existing subgroup.
6. Apply method *PrintItem()* for an existing item and for a non-existing item.
7. Create object containing 30 items and apply method *PrintAll()*.
8. Apply methods *PrintGroup()* and *CountGroupItems()* for a non-existing group.
9. Apply method *InsertItem()* if:
    a. The new item will be a member of an existing group and an existing subgroup.
    b. The new item will be a member of an existing group but the subgroup does not exist.
    c. The new item will be a member of a non-existing group.
    d. This item already exists.
    e. Apply method *PrintAll()* to check the results.
10. Apply method *InsertSubgroup()* if:
    a. The new subgroup will be a member of an existing group.
    b. The new subgroup will be a member of a non-existing group.
    c. The subgroup already exists.
    d. Apply method *PrintAll()* to check the results.
11. Apply method *InsertGroup()* if:
    a. The group is new.
    b. The group exists.
    c. Apply method *PrintAll()* to check the results.
12. Create object containing 100 items and apply method *PrintAll()*.
13. Apply method *RemoveItem()* so that:
    a. After removing the subgroup does not disappear.
    b. After removing the subgroup disappears but the group is kept.
    c. After removing the group disappears.
    d. The item to remove does not exist.
    e. Apply method *PrintAll()* to check the results.
14. Apply method *RemoveSubgroup()* so that:
    a. After removing the group is kept.
    b. After removing the group disappears.
    c. This subgroup does not exist.
    d. Apply method *PrintAll()* to check the results.
15. ApplyMethod RemoveGroup() so that:
    a. The group was an existing one.
    b. The group does not exist.

c. Apply method *PrintAll()* to check the results.

Tips

To get subgroups containing more than one item set the total number of items 200 or greater.

Appendix: printout example

```
A:
2: Great Crested Grebe 04 Oct 2018

C:
26: Moorland Francolin 26 Oct 2018

E:
86: Desert Finch 02 Jun 2018

H:
34: Ruby-throated Hummingbird 12 Nov 2018

I:
9: Sooty Tern 27 May 2018
26: Cuckoo 06 Oct 2018

J:
2: Summer Tanager 05 Oct 2018
43: Iraq Babbler 02 May 2018
60: Erckel's Francolin 13 Jul 2018

M:
26: Ural Owl 23 Apr 2018
44: Grey Francolin 27 Jul 2018

N:
```

85: Blue Rock Thrush 15 Aug 2018

P:

4: Red-breasted Merganser 08 Mar 2018

35: Krüper's Nuthatch 27 Mar 2018

Q:

19: Mute Swan 12 Feb 2018

T:

93: Nubian Nightjar 15 Jun 2018

V:

20: White-backed Woodpecker 04 Aug 2018

X:

6: Ring-necked Duck 26 Apr 2018

60: Small Button Quail (Andalusian Hemipode) 26 Mar 2018

Y:

45: Sooty Falcon 16 Jan 2018