ASSIGNMENT 4

Subject : Binary Search Tree
TAs: [redacted]
Programming Lanuage: C++
Due Date: 25.12.2022

1 Introduction

In this assignment, you are expected to create a two-phase binary search tree (TPBST) structure, in which each node of main Binary Search Tree (BST) keeps a pointer to its own uniquely associated secondary BST. You will use this specialized two-phase binary search tree structure as the in-memory database of a webstore that has different categories and items belongs to categories.

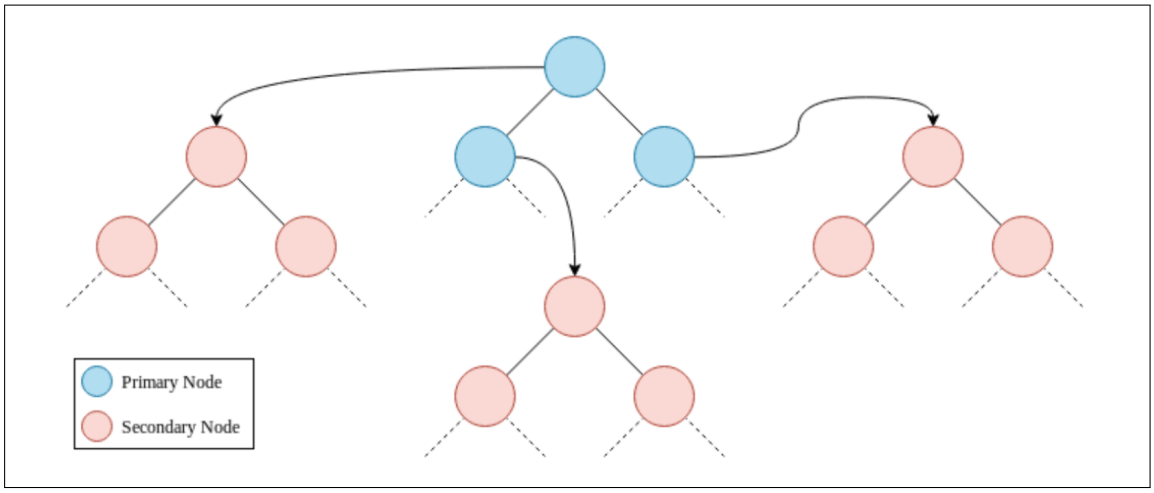


Figure 1: Two-Phase Binary Search Tree structure

2 Background

In Figure 1, the tree consisting of the blue nodes (called primary nodes) is the main tree of the TPBST structure. Each of those primary nodes includes a key, two pointers to its left and right primary node children, and also a pointer to its auxiliary secondary tree which consists of secondary nodes. A secondary node basically includes a key, two pointers to its left and right secondary node children, and also the data. The structure of secondary nodes can also have different variables depending on the first and second part. For example, in the red black trees each node has an extra bit, and that bit is often interpreted as the color (red or black).

To illustrate the structure, one can think of the primary nodes in the main tree as ”categorization” nodes where any item that belong to this category are added into that node’s secondary BST structure as secondary nodes, where they are ordered using a unique key - called the secondary key. Primary nodes of the main tree are also ordered among themselves based on another unique key called the primary key. A secondary tree will not include nodes - say Node A and Node B - with the same secondary keys; however, another secondary tree that belongs to another primary

node (category) can include an element that has the same secondary key with Node A and Node B.

2.1 AVL Trees

AVL tree is a height-balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree. A binary tree is said to be balanced if, the difference between the heights of left and right subtrees of every node in the tree is either -1, 0 or +1. In other words, a binary tree is said to be balanced if the height of left and right children of every node differ by either -1, 0 or +1. In an AVL tree, every node maintains an extra information known as balance factor.

```
Balance factor = height Of Left Subtree - height Of Right Subtree
```

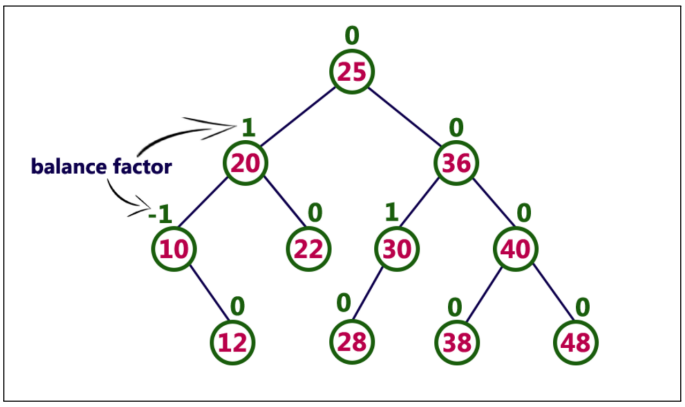


Figure 2: AVL tree example

The tree shown in Figure 2 is a binary search tree and every node is satisfying balance factor condition. So this tree is said to be an AVL tree. To balance itself, an AVL tree may perform the following four kinds of rotations:

- Left rotation
- Right rotation
- Left-Right rotation
- Right-Left rotation

The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2.

2.2 Red Black Trees

A red-black tree is a kind of self-balancing binary search tree where each node has an extra bit, and that bit is often interpreted as the color (red or black). These colors are used to ensure that the tree remains balanced during insertions and deletions. A left leaning Red Black Tree or (LLRB), is a variant of red black tree, which is a lot easier to implement than Red black tree itself and guarantees all the search, delete and insert operations in O(logn) time.

Characteristics of LLRB:

1. Root node is always BLACK in color.
2. Every new Node inserted is always RED in color.
3. Every NULL child of a node is considered as BLACK in color.

4. There should not be a node which has RIGHT RED child and LEFT BLACK child(or NULL child as all NULLS are BLACK) if present , left rotate the node, and swap the colors of current node and its LEFT child so as to maintain consistency for rule 2 i.e., new node must be RED in color.
5. There should not be a node which has LEFT RED child and LEFT RED grandchild, if present Right Rotate the node and swap the colors between node and it's RIGHT child to follow rule 2.
6. There should not be a node which has LEFT RED child and RIGHT RED child, if present invert the colors of all nodes i. e., current_node, LEFT child, and RIGHT child.

3 Experiment

In this assignment, you are expected to write an application that design a webstore. This application includes two parts given below. Secondary tree which consists of secondary nodes must be self-balancing binary search tree for both part 1 and part 2 **(there is no such a rule for primary tree).**

Following are the conditions for a self-balancing binary search tree :

- difference between the left and the right subtree for any node is not more than one
- the left subtree is balanced
- the right subtree is balanced

You should consider each secondary tree separately. As shown in Figure 3, secondary tree of food category and secondary tree of music category are balanced.

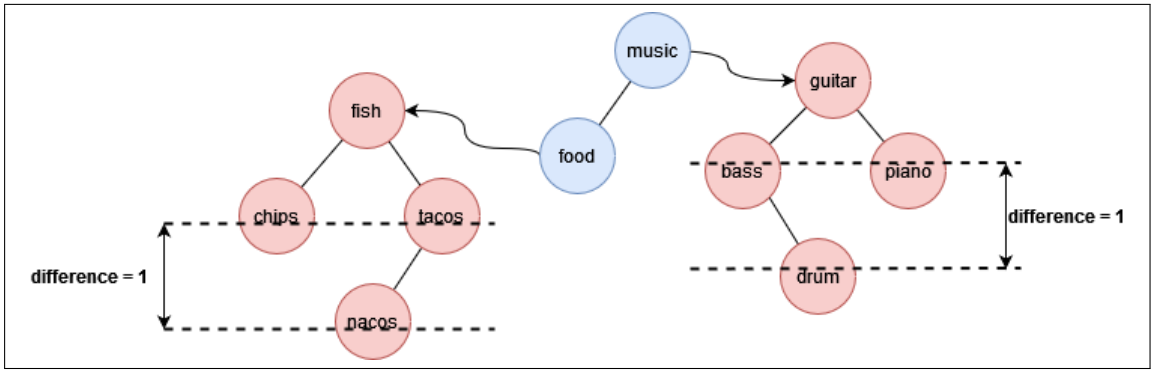


Figure 3: Balanced secondary trees

Balanced and unbalanced trees are shown in Figure 4. As shown in the figure, tree at the top of the picture have unbalanced secondary tree (secondary tree of music node). Because the difference between the left and the right subtree for guitar node is more than one. A balanced version of the this unbalanced secondary tree is shown at the bottom of the figure.

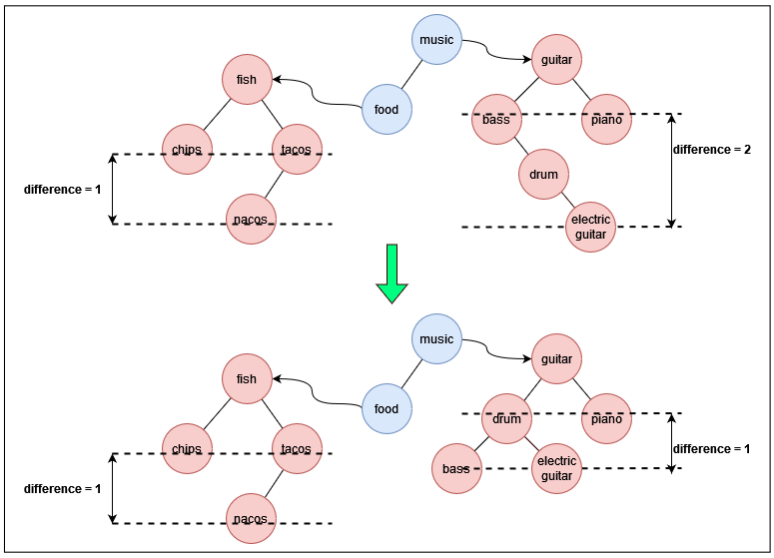


Figure 4: Balanced and unbalanced secondary trees

3.1 First Part

In our Webstore scenario, primary nodes in the main tree will have the categories of the inserted items as keys to apply lexicographical BST ordering among category names; and secondary nodes will use item names, that are unique in that category, as keys to apply lexicographical BST ordering among items in that secondary tree.

PrimaryNode represents nodes that constitute the main tree. A PrimaryNode keeps a key variable which is an instance of string to uniquely identify the node in the main tree, two pointers to its left and right PrimaryNode children, and a pointer to the root of its own secondary BST, which consists of SecondaryNodes.

SecondaryNode represents nodes that constitute secondary trees. A SecondaryNode keeps a key variable which is an instance of string to uniquely identify the node in its tree, two pointers to its left and right SecondaryNode children, and the data variable to hold the data.

Each item has four attributes: category, name, and price. There is no single attribute that uniquely identifies items. However, no two items with the same category and name may exist. All attributes of an item are set during the object construction and all of them other than price can not be changed later.

In the TPBST that will be utilized in the webstore application, category attributes will be used as primary keys, name attributes will be used as secondary keys, and price attributes will be used as data. This overall approach will constitute the database structure of our webstore. By using this approach, we will be able to access and print the details of each item individually; as well as printing all information currently contained in our TPBST. Moreover, the structure will enable us to access and print the details of all items that belong to the category that we are interested in.

There are several commands will be given in input file for this experiment. You will read input file and create an output file for the first part. The commands will be given are as follows:

- `insert<TAB><category><TAB><name><TAB><price>`
- `remove<TAB><category><TAB><name>`
- `printAllItems`
- `printAllItemsInCategory<TAB><category>`

- printItem<TAB><category><TAB><name>
- find<TAB><category><TAB><name>
- updateData<TAB><category><TAB><name><TAB><newPrice>

3.2 insert

Insertions to the secondary tree are expected to be done according to AVL tree.

- If TwoPhaseBST is empty, insert a PrimaryNode created by using the given "category" to the empty tree, then create a SecondaryNode by using the given "name", and make the rootSecondaryNode pointer of the newly created PrimaryNode point to that SecondaryNode. Store the data in the SecondaryNode. **The insertion to the tree should be done in a way that preserves the balance of the tree.**
- If TwoPhaseBST is not empty, search for the node - say Node A - with the key value category among the PrimaryNodes in the main tree.
  - If found, create a SecondaryNode with the key value **name**(secondaryKey), store the data in it, and insert it to the appropriate location of lexicographical order in the secondary tree of Node A. Names are guaranteed to be unique in the secondary tree of a primary node; so, no need to check for duplications.
  - If not found, then create a new PrimaryNode with the given **category**(primaryKey) and insert it to the main tree using lexicographical ordering. Then, create a SecondaryNode with the key value name, store the data in it, and make rootSecondaryNode pointer of Node A point to that SecondaryNode.

  In Figure 5, we have two primary nodes as categories in the main tree, together with their secondary trees, whose nodes use item names as keys. We add a new item with its category being music, and name being harp. We first search the main tree for the key music, then we create the SecondaryNode that keeps the item data, and add that SecondaryNode to its proper place: as the left child of the piano SecondaryNode. Figure 6 shows the resulting TPBST after the insertion operation is completed.

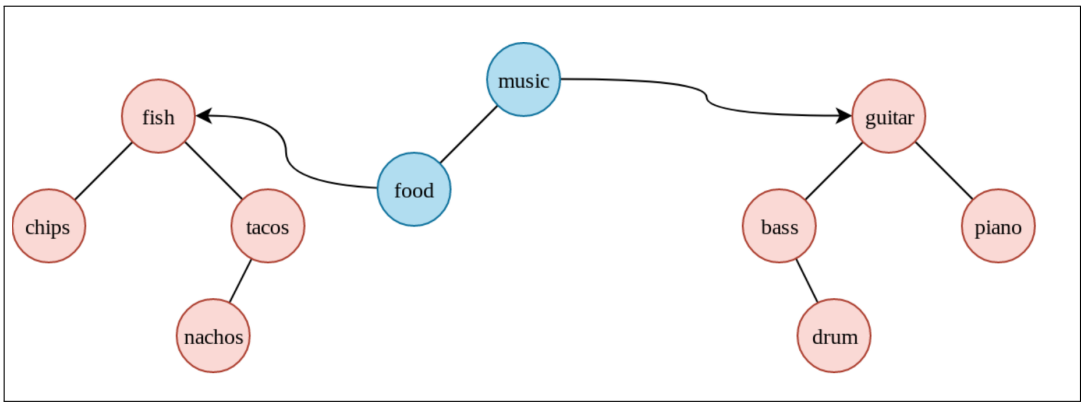


Figure 5: Before the insertion of the item, named ”harp”

In Figure 7, we add a new item with its category being sports, and name being golf. We first search the main tree for the key sports, but it does not exist. So, we first create the sports PrimaryNode, then we create the SecondaryNode that keeps the item data, and make the rootSecondaryNode of the sports node to point to that SecondaryNode. Figure 7 shows the resulting TPBST after the insertion operation is completed.

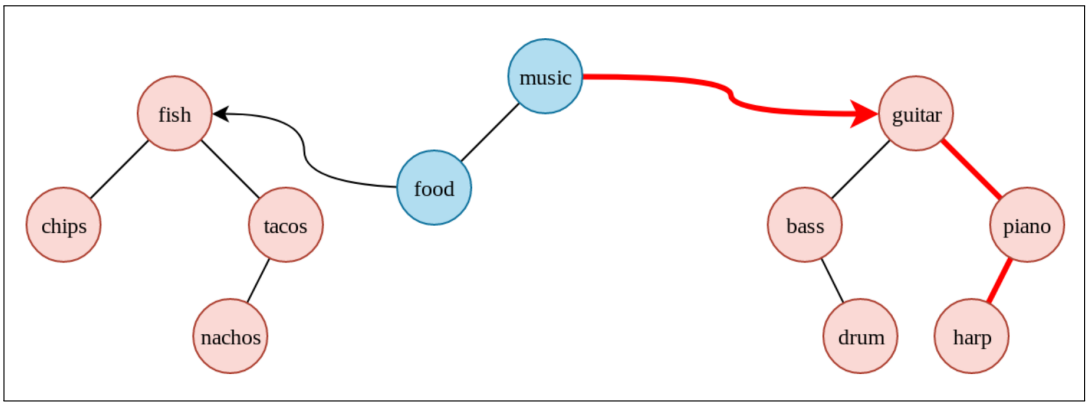


Figure 6: After the insertion of the item, named ”harp”

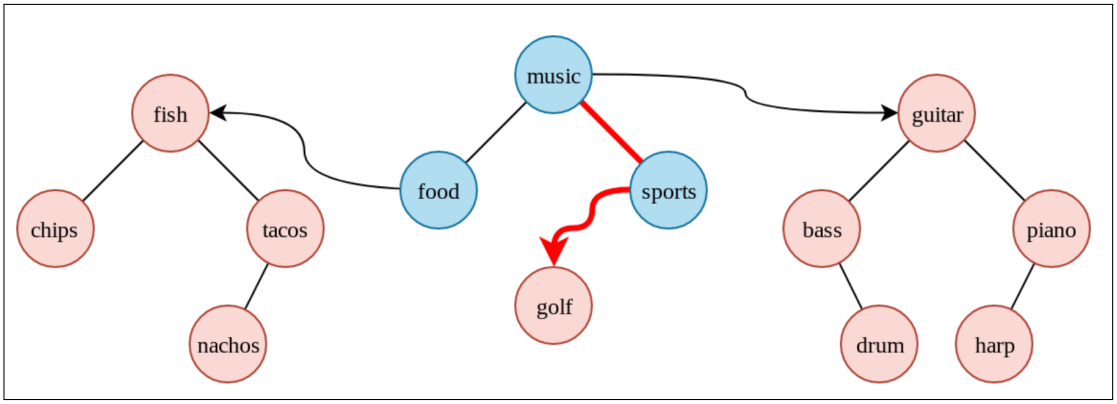


Figure 7: After the insertion of the item, named ”golf”

4 remove

Find the SecondaryNode by using the category to search the main tree, and the name to search the corresponding secondary tree if the PrimaryNode exists. If the SecondaryNode is not found, do nothing. If it exists, remove that SecondaryNode by applying BST deletion procedures to preserve the lexicographical ordering among the nodes of the secondary tree (see Figure 8, then Figure 9). Note that only the SecondaryNodes can be removed; PrimaryNodes cannot be removed once created. In other words, the secondary tree of a PrimaryNode can become NULL if all of its SecondaryNodes are removed; but the PrimaryNode will still exist (see Figure 7, then Figure 8). **As in the case of insertion, you should remove an item according to AVL tree rules and the balance of the tree should be preserved in the deletion process.**

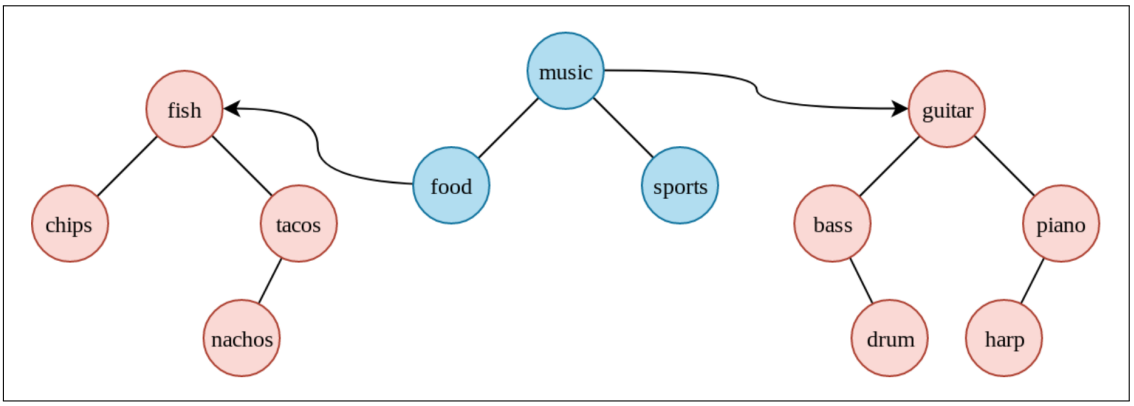


Figure 8: After the removal of the item, named ”golf”

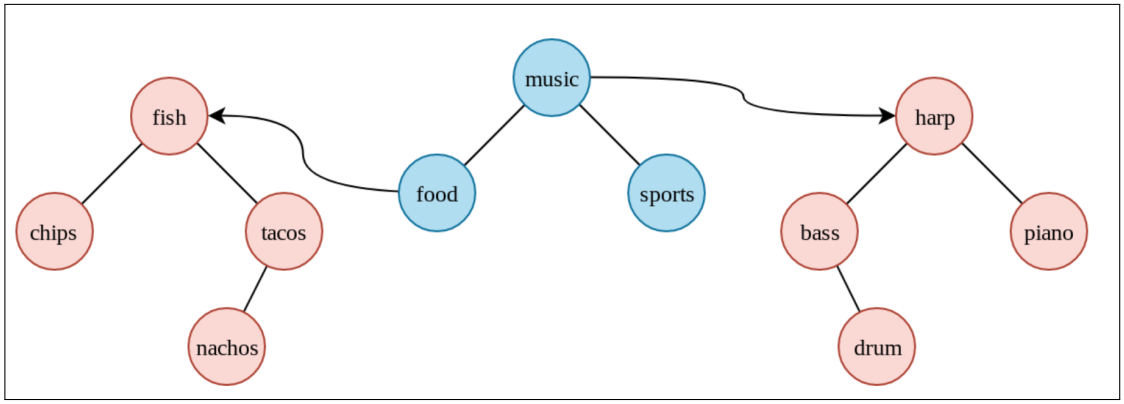


Figure 9: After the removal of the item, named ”guitar”

5 printAllItems

Information about all the items in the tree should be printed. You should print nodes of both primary tree and secondary tree level by level. The strict printing format is as follows:

```
{
 "p_key_1":
    "s_key_11":"data_11"
     .....
 "p_key_2":
    "s_key_21":"data_21", ...
     ....
}
```

In this format, p_key's are primary keys, s_key's are secondary keys, data's are data of items. If the tree is empty (i.e. no primary and secondary nodes), then print as follows:

```
{}
```

If primary key exists in the tree, but it has no secondary nodes, then print as follows:

```
{"p_key":{}}
```

As an example, for the sample tree in Figure 9, the output of the print() function call must be as

follows (data representations may change):

```
{
 "music":
       "harp":"120"
       "bass":"100”,"piano":"350"
       "drum":"115",
 "food":
       "fish":"20”
       "chips":"10","tacos":"15"
       "nachos":"20"
 "sports":{}
}
```

Note that both primary tree and secondary trees printed level by level. As shown in Figure 9, music,food and sports be printed respectively to print the primary tree level by level. And the secondary tree of each primary tree should be printed level by level in the same way.

6 printAllItemsInCategory

Print information that belongs to all item objects in the tree with the given category parameter.

The strict printing format is as follows:

```
{
 "p_key_1":
    "s_key_11":"data_11"
    "s_key_12":"data_12", ....
     .....
}
```

7 printItem

Print information of a single item object that is specified by the given category and name parameters. The strict printing format is as follows:

```
{
 "p_key":
    "s_key":"data"
}
```

8 find

Search the main tree with the given category to find the PrimaryNode that we are interested in, and then search its secondary tree with the given name. If the SecondaryNode is found, print the data stored in that SecondaryNode like in Section 7. If the PrimaryNode or the SecondaryNode do not exist, then print {}.

8.1 Second Part

As a second part of the assignment, insertions to and deletions from the secondary tree are expected to be done according to Left Leaned Red-Black Tree (LLRBT). Left-leaning red black tree details

are given in Section 2.2. This part will be test same commands stated in Section 3.1. You will read input file and create an output file for the second part. The commands in this part are the same as in the first part of assignment. The only difference is in the implementation of insertions and deletions of secondary tree, so there may be differences in the output of the print command.

9 Inputs and Outputs

For this assignment you have one input file that contains commands. You are expected to produce two output file for both first part and second part according to input file. The format of input and output files are as shown in Figure 10 and Figure 11. Please create your output file according to the format given to you (Your output file format should be as shown in Figure 11). Note that there are some differences between the output of the first part and the second part as shown in Figure 11. Because in the first part the insertions and deletions were done according to the AVL tree, while in the second part they were done according to the LLRB.

```
insert music guitar 150
insert music bass 200
insert food fish 30
insert food tacos 60
insert food nachos 55
insert music drum 250
insert music piano 300
printAllItems
updateData music bass 250
printItem music bass
insert food chips 45
insert food chicken 60
insert sports dumbbell 200
remove sports dumbbell
printAllItemsInCategory sports
insert clothes sweatshirt 100
insert clothes socks 25
insert clothes jean 125
insert clothes tie 35
printAllItemsInCategory clothes
remove clothes sweatshirt
printItem clothes sweatshirt
printAllItems
find music electric guitar
find clothes tie
```

Figure 10: input.txt

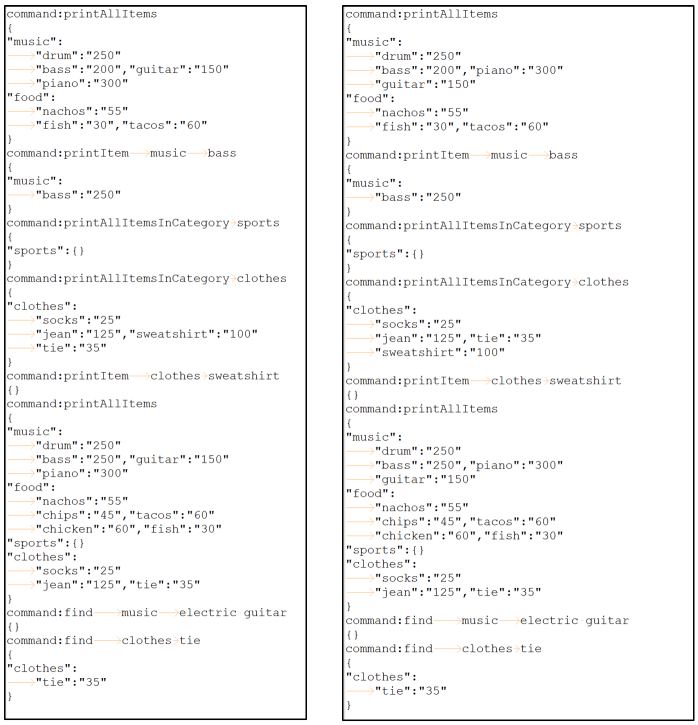
10 Execution

The name of the compiled executable program should be "webstore". Your program should read input/output file names from the command line, so it will be executed as follows:

```
webstore [input file name] [output file to 1st part] [output file to 2nd part]
```

```
./webstore input.txt first_part_output.txt second_part_output.txt
```

You can see sample input and output files in piazza page. The program must run on DEV (dev.cs.hacettepe.edu.tr) UNIX machines. So make sure that it compiles and runs on dev server. If we are unable to compile or run, the project risks getting zero point. It is recommended that you test the program using the same mechanism on the sample files (provided) and your own inputs. You must compare your own outputs and sample outputs. If your outputs are different from the sample, the project risks getting zero point, too.



output1.txt

```
command:printAllItems
{
"music":
	"drum":"250"
	"bass":"200","guitar":"150"
	"piano":"300"
"food":
	"nachos":"55"
	"fish":"30","tacos":"60"
}
command:printItem	music	bass
{
"music":
	"bass":"250"
}
command:printAllItemsInCategory sports
{
"sports":{}
}
command:printAllItemsInCategory clothes
{
"clothes":
	"socks":"25"
	"jean":"125","sweatshirt":"100"
	"tie":"35"
}
command:printItem	clothes sweatshirt
{}
command:printAllItems
{
"music":
	"drum":"250"
	"bass":"250","guitar":"150"
	"piano":"300"
"food":
	"nachos":"55"
	"chips":"45","tacos":"60"
	"chicken":"60","fish":"30"
"sports":{}
"clothes":
	"socks":"25"
	"jean":"125","tie":"35"
}
command:find	music	electric guitar
{}
command:find	clothes tie
{
"clothes":
	"tie":"35"
}
```

output2.txt

```
command:printAllItems
{
"music":
	"drum":"250"
	"bass":"200","piano":"300"
	"guitar":"150"
"food":
	"nachos":"55"
	"fish":"30","tacos":"60"
}
command:printItem	music	bass
{
"music":
	"bass":"250"
}
command:printAllItemsInCategory sports
{
"sports":{}
}
command:printAllItemsInCategory clothes
{
"clothes":
	"socks":"25"
	"jean":"125","tie":"35"
	"sweatshirt":"100"
}
command:printItem	clothes sweatshirt
{}
command:printAllItems
{
"music":
	"drum":"250"
	"bass":"250","piano":"300"
	"guitar":"150"
"food":
	"nachos":"55"
	"chips":"45","tacos":"60"
	"chicken":"60","fish":"30"
"sports":{}
"clothes":
	"socks":"25"
	"jean":"125","tie":"35"
}
command:find	music	electric guitar
{}
command:find	clothes tie
{
"clothes":
	"tie":"35"
}
```

Figure 11: Output files

11 Grading and Evaluation

- Your work will be graded over a maximum of 100 points.
- Your total score will be partial according to the grading policy stated below.

Compiled	5p
Insert (1st part & 2nd part)*15	30p
Remove (1st part & 2nd part)*15	30p
printAllItems	6p
printAllItemsInCategory	6p
printItem	6p
find	6p
updateData	6p
Code design, clean and readable code, algorithmic perspective, comments	5p

- **Your code will be tested on dev platform with different inputs. And one output file will be expected as output file. If your program cannot compile on dev server, you cannot get any points. Please be sure that your program can be compile on dev server.**
- You should create and submit a ZIP archive in the following structure for evaluation. An invalid structured archive will cause you partial or full score loss. You are required to submit a Makefile, which will be used to compile your program.
- **Assignments that do not produce any output file will not be evaluated.**

Notes

- Do not miss the deadline.
- Save all your work until the assignment is graded.
- The assignment must be original, individual work. Duplicate or very similar assignments are both going to be considered as cheating.
- Write READABLE SOURCE CODE block
- You can ask your questions via Piazza (https://piazza.com/hacettepe.edu.tr/fall/bbm203) and you are supposed to be aware of everything discussed in Piazza.
- You will use online submission system to submit your experiments. https://submit.cs.hacettepe.edu.tr/ No other submission method (email or etc.) will be accepted. Do not submit any file via e-mail related with this assignment.
- File hierarchy must be zipped before submitted (Not .rar, only .zip files are supported by the system). You must submit your work with the file hierarchy stated below:

<student_id.zip>/(Required)

→src/(Required)

→*.cpp

→*.h

→Makefile (Required)

Policy

All work on assignments must be done **individually** unless stated otherwise. You are encouraged to discuss with your classmates about the given assignments, but these discussions should be carried out in an **abstract** way. That is, discussions related to a particular solution to a specific problem (either in actual code or in the pseudocode) **will not be tolerated**. In short, turning in someone else's work (from internet), in whole or in part, as your own will be considered **as a violation of academic integrity**. Please note that the former condition also holds for the material found on the web as everything on the web has been written by someone else.

References

https://read.seas.harvard.edu/ kohler/notes/llrb.html

https://www.programiz.com/dsa/avl-tree

https://www.programiz.com/dsa/red-black-tree