Task 3

Final Submission Date - 20.01.2021

Caution: Do not copy the program output messages from the job's speech. In in these cases non-ASCII characters are entered, causing the script running in autolab to terminate unexpectedly and without sufficient justification and not output the actual differences between your output and the desired one.

Required Data Structures of the Job

In order to carry out this work, it is necessary to implement the data structures described below. These structures store integers of type int (or long int) or whatever data type you think suits you. Each structure is implemented by two files, one with a .h extension and one with a .c extension. The files with the ending .h contain the structs of each data structure and the prototypes of the functions that implement it, and in .c only the implementations of the corresponding functions.

The binary search tree

In the files**tree.c, tree.h** implement a binary search tree.

Binary Trees 1

In this exercise you will write a program that implements the following variant of AVL trees, which will be called HBTree (Height Balanced Tree). A node **K**of a binary tree of searchHBTree is considered balanced if the height difference between the left and right of its subtree is less than or equal to **max(1,⌊ log ₂(N)⌋)** , where **N** the number of subtree nodes rooted at node **K**.

The HBTree tree introduces, in relation to the AVL tree, an additional field at each node which is its weight and represents the number of nodes contained in the subtree rooted at this node.

The kinds of rotations performed to balance the tree are the same as the kinds of rotations of an AVL tree. The tree balancing methodologies during node insertion and deletion are similar to the corresponding methodologies for AVL trees. More specifically:

- When inserting a new node we ascend from the new node to the root updating the heights and weights of each node in the path. If the HBTree condition of the tree is violated at any node on the path we apply the appropriate kind of rotation, The operation is terminal.

- When deleting a node we ascend from the parent of the deleted node to the root updating the heights and weights of each node along the path. If the tree condition is violated for any node on the path we apply the appropriate kind of rotation and examine whether the nodes involved in the rotation are balanced based on the HBTree condition. If any node that participated in the rotation is not balanced after it, we must recursively restore the balance with top node that node. The act of rotation is not terminal. We continue the process until we reach the root.

The structs and prototypes of the HBTree methods will be in the file hbtree.h and the implementations of its methods in the file hbtree.c .

Write the program tree1.c which reads from standard input a mixed sequence of nonnegative integers and commands. The commands that can be read are the following:

- **- i (insert):** nesting of all integers following the given command and up to read new command (except command - **p**). For each number that is attempted to be entered into the tree, a newline character is printed, the alphanumeric " INSERTED X " if the insertion was successful or " NOT INSERTED X " if the insertion was not successful, where X is the number that was attempted to be inserted into the tree. A newline character is printed at the end.
- **- d (delete):** delete all integers following this command and up to read new command (except command - **p**). For each number that is attempted to be deleted from the tree, a newline character is printed, the alphanumeric " DELETED X " if the deletion was successful or " NOT DELETED X " if the deletion was not successful, where X is the number that attempted to be deleted from the tree. A newline character is printed at the end.
- **- f (find):** search for all integers following the given command in the tree and until a new command is read (except the command - **p**). For each integer searched for, a newline character is printed, the alphanumeric " FOUND X " if the search was successful or " NOT FOUND X " if the search was not successful, where X is the number that sought in the tree. A newline character is printed at the end.

- **- p (print):** printout of the tree. A newline character is printed first, at then the contents of the tree during pre-order penetration and at the end again a newline character. For tree contents, each integer is followed by a blank character.

- **- q (quit):**terminates the program.

  The insertion of this order does not negate the effect of the order that preceded it, with respect to the numbers that follow. For example, the sequence **-i 10 5 20 -p 33 48** initially it will

enters the numbers 10 5 20, then it will print the tree and finally it will enter the numbers 33 48.

Question: A UK treeDoes Tree have logarithmic height? What is the maximum number of rotations when inserting and deleting a node of the tree?

Binary Trees 2

Write the programtree2.c , which reads a sequence of positive integers from standard input (*stdin*) , which constitutes thepre-order traversal of a binary search tree. The numbers in the sequence are separated by one or more spaces, and the sequence ends by reading a negative number. The program constructs the binary search tree corresponding to the given penetration.

The program then checks whether the tree read can be a red-black tree. If the tree read represents a feasible red-black tree then it prints the message "RB OK" , while otherwise it prints " RB NOK " . Next is the printing of two characters line change.

In case the tree can be red-black, it prints the pre-order penetration as a red-black tree. For each node, the corresponding integer, '#' character (pound), and 'B' character if the node is black or 'R' character if the node is red are printed. Each node is printed after a blank character and the last blank is followed by a newline character.

Fragmentation

Write a program that implements a hash table with open or free addressing and a linear test with respect to the placement function. The program uses reshuffling to keep the load factor within the desired range. The array stores pointers to positive integers. An empty location contains the value 0 NULL , while a vacated (location whose content has been erased) contains the value (int*)0x1 -1 .

Rehashing works like this:

- After ~~-~~ Before an insert if the load factor is greater than or equal to 0.5 the array ~~-~~ doubles its size and the elements are rearranged into it.
- After a deletion if the load factor is less than or equal to 0.125 the table is ~~halv~~ed and the elements are placed back into it.

The structs and prototypes of the hash table methods will be in the file htable.h and the implementations of its methods in the file htable.c.

Write the program**htable1.c** which reads from the standard input a scrambled sequence of commands and positive integers. The commands that can be read are the following:

- **- i (insert):** nesting of all integers following the given command and up to read new command (except command - **p**). For each number that is attempted to be entered into the table, a newline character is printed, the alphanumeric " INSERTED X " if the insertion was

successful or " NOT INSERTED X " if the insertion was not successful, where X is the number that attempted to enter. A newline character is printed at the end.

- **- d (delete):** delete all integers following this command and up to read new command (except command - **p**). For each number that is attempted to be deleted from the table, a newline character is printed, the alphanumeric " DELETED X " if the deletion was successful or " NOT DELETED X " if the deletion was not successful, where X is the number that attempted to be deleted. A newline character is printed at the end.
- **- f (find):** search for all integers following the given command in the array and until a new command is read (except the command - **p**). For each integer searched for, a newline character is printed, the alphanumeric " FOUND X " if the search was successful or " NOT FOUND X " if the search was not successful, where X is the number that was searched for. A newline character is printed at the end.

- **- p (print):** print the table.
  - A line break character and the alphanumeric " are initially printedSIZE: S INSERTED: X, DELETED: Y " followed by a newline character, whereS the size of the board, X the number of occupied board slots, and Y the number of vacated board slots that contain tombstones.

  - Then all positions of the array are printed starting the traversal from the first position (position 0) to the last (position S-1) with a width of three decimal places followed by a blank character. After the last space, a newline character is printed.

  - Finally, the contents of all occupied table positions are printed, starting with the penetration from the first position (position 0) to the last (position S-1).
    - If the position is occupied, the integer contained in it is printed with width of three decimal places followed by a blank character.
    - If the position is empty two spaces, an asterisk '*' and a space are printed.
    - If the location contains a tombstone, two spaces, a '#' punch and a space are printed.
- **- r (rehashing):** If the rehashing function is active, it also stops being active array stops resizing based on the values of *load factor*. If the rehashing function is disabled, it is enabled.

- **- q (quit):**terminates the program.

The program always starts from an initial table size equal to 2 which is also the minimum possible size for the table. When starting the program the defragmentation mode is active.

Shipping Method

The work will be sent through the platform autolab (no VPN connection required). Follow these steps:

- Create a folder named **hw3submit**and copy the files in there**hbtree.c, hbtree.h, tree1.c, tree.c, tree.h, tree2.c, htable.h, htable.c, htable1.c** .
- Compress as tar.gz. On Linux/KDE go to the folder, right click and select **Compress -> Here (as tar.gz).**The file is created**hw3submit.tar.gz** .
- You connect to the address **https://autolab.e-ce.uth.gr** and choose the course **ECE215-F20 (f20)** and from this work**HW3** .

- To submit your work, click on the option **"I affirm that I have compiled with this course academic integrity policy..."**and press**submit** . Then select the file **hw3submit.tar.g** z that you created in step 2.

Appendix - Print tree in image

To print a tree (or a graph) in an image you can use the program dot of the suitegraphviz . The dot program can draw the tree or graph in any image format (png, jpg etc) or in pdf format if it receives as input a suitable text file containing the information of the tree or graph.

For each tree you want to paint, it is enough to programmatically construct the corresponding text file (let's call it a dot file). The name and extension of the file don't matter, but by convention let's give it the extension dot.

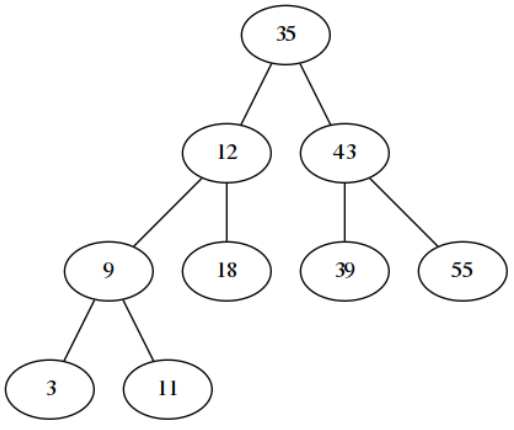
The contents of the file are as follows. In principle, the information is enclosed in brackets as below:

```
graph G {
   /* here the content of the graph or tree */
}
```

The order in which the nodes appear in the file does not really matter. But it is good that any tree is printed during the pre-order penetration, so that the root appears at the top. Each node has a unique identifier, this can be anything. For simplicity, set each node's unique identifier to the integer stored inside it. Finally, remember that it doesn't really matter if the dot file shows a node's connections first and then the node itself.

Let's look at the example tree below, where the dot file is listed on the left and the resulting image on the right. Notice that the connections of nodes (tree edges) are denoted by two attached forward slashes (--) between the unique identifiers of the individual nodes. Spaces before and after dashes are optional and help make the file readable.

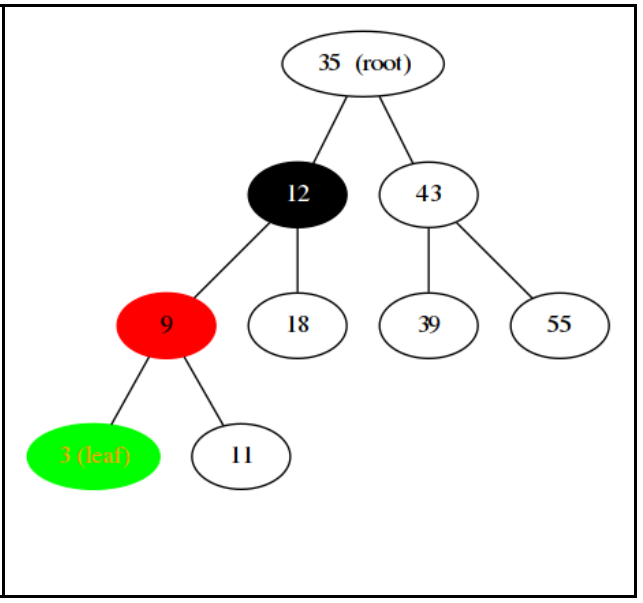
```
graph  G {
   35
   35 -- 12
   12
   12 -- 9
   9
   9 -- 3
   3
   9 -- 11
   11
   18
   12 -- 18
   35 -- 43
   43
   43 -- 39
   39
   43 -- 55
   55
}
```



In each node, you can have additional information about how you want it to appear. This information is placed immediately after the identifier of each node in brackets. If you do not add relevant information, each node appears with a white circle with a black one that contains the information inside it in black letters (as in the previous figure).

Below we rewrite the dot file so that in some nodes we can change the background color or add additional information that we want to appear on the node or change their identifier.

```
graph G {
   theroot     [label="35         (root)"]
   theroot     - - 12
   12  [style="filled",color="black",fontcolor="white"]
   12  - - 9
   9  [style="filled",color="red",fontcolor="black"]
   9  - - 3
   3  [style="filled",color="green",fontcolor="orange"
label="3      (leaf)"]
   9  - -   11
   11
   18
   12  - -   18
   theroot     - - 43
   43
   43  - -   39
   39
   43  - -   55
   55
}
```



Changes:

1. Tree root changed unique identifier. From 35 it became **theroot** . No problem as long as this replaces all occurrences of 35 as a unique identifier.
2. Nodes 12, 9 and 3 changed color.
3. In the root node (35) and in node 3 we also changed the label shown in the image.

From the dot file in the image

As long as we have constructed the dot file correctly the transform to image is simple. Just run the command below (in yellow) from the command line.

 # > dot -T png FILE.dot -o FILE.png

where FILE.dot is the name of the dot file and FILE.png is the name of the resulting file in png format. The dot and png files should not have the same name, but we follow "consistent" nomenclature to know which dot file corresponds to each png file. If you don't like png format and want jpg or pdf you can write.

 # > dot -T jpg FILE.dot -o FILE.jpg **the** **# > dot -T pdf FILE.dot -o FILE.pdf**

Programmatic implementation of the transformation

If you want to programmatically implement the above transformation in C, you can do it with the help of what you learned in the Programming II lesson. First, you create a new child process from the parent process. Through the new process, run the dot program with the parameters you want. The parent process should wait for the child process to finish processing before continuing.

Alternatively you can use the system (man system) function, which the shell uses to execute any command.