Task #7

Balanced trees, hash tables

The aim of the work is to construct and process hash tables, compare the efficiency of search in balanced trees, in binary search trees and in hash tables. Compare the efficiency of collision elimination in external and internal hashing.

Assignment:

Build a hash table using the specified data. Compare the efficiency of searching in a balanced binary tree, in a binary search tree and in a hash table (using open and closed addressing). Display the trees and the hash table on the screen. Calculate the average number of comparisons for searching data in the specified structures. Restructure the hash table if the average number of comparisons is greater than the specified number. Evaluate the efficiency of using these structures (in terms of time and memory) for the task at hand. Evaluate the efficiency of hash table search at different number of collisions and at different methods of their resolution.

No. of variant - No. of list % 7

0. Balance the tree (task #6) after removing repeated letters. Display it on the screen as a tree. Make a hash table containing letters and the number of their occurrences in the entered string. Display the table on the screen. Search for the entered letter in a binary search tree, in a balanced tree and in a hash table. Compare the search time, memory size and number of comparisons when using different data structures.

1. Using the previous program (task #6), balance the obtained tree. Display it on the screen as a tree. Build a hash table of the file numbers. Search for the entered integer in the binary search tree, in the balanced tree and in the hash table. Compare the search time, memory size, and number of comparisons using different data structures.

2. Using the previous program (task #6), balance the obtained tree. Display it on the screen as a tree. Build a hash table from the words of the text file, specifying the size of the table from the screen. Search for the entered word in the binary search tree, in the balanced tree and in the

hash table. Compare the search time, memory size and number of comparisons when using different data structures.

3. Build a search tree from the words of a text file, balance the resulting tree. Display it on the screen as a tree. Delete all words beginning with the specified letter in the original and balanced tree. Compare the deletion time and memory size. Build a hash table of words in the text file. Display the constructed table of words on the screen. Search and delete the entered word, output the table. Execute the program for different table sizes and compare the deletion time, memory size and number of comparisons when using balanced trees and hash tables.

4. Build a hash table for C++ reserved words (at least 20 words) containing HELP for each word. Display a hint for the entered word on the screen. Execute the program for different table dimensions and compare the search time and number of comparisons. Create a balanced tree for the specified data. Add a hint for the newly entered word, using table restructuring if necessary. Compare the effectiveness of adding a clue to the table or restructuring the table for different table populations.

5. Build a hash table for words of a text file. Search for the specified word in the binary search tree (BST) and in the hash table; if it is not present, add it (at user's request) to the tree and, accordingly, to the table. Use table restructuring if necessary. Balance the tree. Compare the search time, memory footprint, and number of comparisons when using DDPs, balanced trees, and hash tables. Compare the efficiency of adding a key to the table or restructuring the table for different degrees of table population.

6. Using the previous program (task #6), build a tree, for example, for the following expression: 9+(8*(7+(6*(5+4)-(3-2))+1)). In postfix traversal of the tree, compute the value of each node and write the result to its node. Obtain an array using infix traversal of the resulting tree. Construct a binary search tree (BST) for this data, balance it. Build a hash table for the values of this array. Search for the specified value. Compare the search time, memory size and number of comparisons when using DDP, balanced trees and hash tables.