Slide 1.

Hello everyone. That's a sixth laboratory work for algorithms and data structures. Our topic is data structures.

Slide 2.

Today we will analyze possible solutions of problem "Disk Tree" from your previous homework, problem "White Streaks" and will have small overview of third test's tasks

Slide 3.

Problem 1067 - "Disk Tree". You can go through the link to check full description. I will remind you the most important part

- Bill has several copies of directory listings from his hard drive. Using those listings, he was able to recover full paths (like "WINNT\SYSTEM32\CERTSRV\CERTCO~1\X86") for some directories. He put all of them in a file by writing each path he has found on a separate line.
- Your task is to write a program that will help Bill to restore his state of the art directory structure by providing nicely formatted directory tree.

Slide 4.

Important note for this problem is that all paths are started from the root and contains only directories.

The first question for this problem: what is file system? The most obvious way to present file system is a tree, and nodes of this tree are directories.

Slide 5.

Let's take example from the problem's description and present it in a tree. We will add a root node to present starting directory. Then split each path by slash and add its parts to the tree if they are absent in the tree. First part of path will be added as child of root (if absent), second part will be added as child of first part's node, and so on.

For example, let's add path WINNT\SYSTEM32\ALGORITHMS to the tree in the slide. Firstly, we will check, if root has child with name WINNT. On the next step we can "open" directory WINNT and check, is it have directory SYSTEM32. Then we can "open" this directory and check for directory ALGORITHMS. This directory is absent, so we should add it as child of directory SYSTEM32.

Slide 6.

How directory can be present in the program? Let's check a subtree. Should be noted, that each directory can have multiple directories inside it. So, first suggestion is that each node should contain linear data structure like array, vector or list to store its children.

But how to identify specific child? It is obvious, that each node has a name, which can be used to identify it among its siblings. So suggested array should contain pairs of directory name and another vector of its children. This structure looks like map.

Slide 7.

So, the key to solve this problem is creating recursive data structure, so each element of this structure's type should contain map, where key is names of children directories, and value is the same recursive data structure. For leaves map should be empty.

Slide 8.

Now we have explanation of data structure and can start to use it. First of all, we should create a tree. For this purpose, we can split each line by slash and insert path to the tree. Here we can get another benefit of using map. On each step we should check, is this directory exist inside parent's map. And for the map search operation is much faster than, for example, for vector of pairs. If directory is absent, so we can add it to the related parent's map.

After the tree is created, it only remains to print all data. It can be realized as traversal of the tree (in similar way with DFS algorithm). On each step we should iterate on map's elements of current directory. Here should be noted, that for map (not hash map) all values are sorted during iterating over map. It is another very useful feature of map for this problem, because we should print directories inside one directory in lexicographic order. Each time we get a directory, we should open it and iterate over its subdirectories.

Number of starting spaces for printing name of directory depends on current depth of recursion. For example, for directories inside root node, they have zero starting spaces, for next level – one starting space and so on.

Slide 9.

Solution mentioned earlier is based on using of data structure. But it is also possible to solve this problem with sorting. Let's quickly check this approach.

First of all, we should convert all paths, as it is mentioned on the slide, and add them to the array. Path should be splitted by slash, then only first word added, after this first two words splitted by space, and so on. Using of space is important for this method.

Then we should sort all created lines. Now they're ordered lexicographically. That's why using of space is important. It is character with least number in ascii table of symbols, so string which starts with WINNT with space will be presented in array earlier, then string which starts with WINNT and any other available symbol.

After this we can remove all duplicates (it is a fast operation, because array is sorted, and all duplicates on the neighboring positions). The last part is printing. For each line last word should be printed. Numbers of starting spaces obviously depends on number of words in the string.

Slide 10.

Problem 1628 - “White Streaks”. You can go through the link to check full description. I will remind you the most important part

- The Martian has a calendar in the form of an m × n table; he marks in this calendar days when he had bad luck. If he had bad luck in the j^{th} day of the i^{th} week, he paints the cell (i, j) black. Initially, all cells are white.
- Let rectangles of the form 1 × l or l × 1 be called segments of life. Maximal with respect to inclusion white segments are called white streaks. It is required to determine how many white streaks there were in his life.

Slide 11.

For this problem we have a table (or two-dimensional array) to represent calendar. There is no need to represent calendar in such way in a program, we will use this format for better demonstration. Our task is to calculate number of white streaks, which is a continuous line of white squares: horizontal or vertical. We will call horizontal lines of this table as rows, and vertical as columns. It is obvious that number of white streaks in rows and number of white streaks in columns can be calculated separately.

Depends on details of algorithm it can be required to add special border cells, which have grey color in image and behaves in the same way as black cell. The main idea is to have strict borders for calculating lengths of white streaks, but in program it can be handled with check, that all cells of streak are either in the same row or in the same column. Usefulness of these borders will be explained later.

Slide 12.

How to find streaks effectively?

Let’s check two black cells: c_1 and c_2. We already have 3 questions to their position. Are they on the same line? If not in the same, so there is no white streak between them. Does they have white cells between them? If there are no white cells, so it isn’t a white streak. Is there any black cell between those two cells? Because if there is any black cell c_3 between them, so this two points has no white streak between them, but it is possible to have white streaks between c_1c_3 or c_2c_3. And such checks should be provided for each pair of black cells.

Slide 13.

But as was mentioned earlier, we can calculate streaks in rows and columns separately. Let’s start from rows. We can sort black cells by rows and columns as secondary parameter. Here we have a benefit from introducing grey cells. In result, which you can see on the slide, grey cells help to split streaks, which should be on different lines.

Now it is easy to calculate all white streaks in rows, just by calculating difference between consequent black and grey cells column positions.

Slide 14.

In the same way we can calculate all white streaks in columns, using sorting by columns and rows as secondary parameter. Is it enough to solve this problem?

Slide 15.

Let’s check the red cell in this example. For fifth column it is a part of white streak, which has size two. But for second row it is a streak of size 1 on 1. So, we will calculate it twice, but it is wrong by description.

1 on 1 streak shouldn’t be calculated in this case, since it is completely included in larger streak.

Slide 16.

Possible naive solution is just to skip 1 on 1 streaks for calculating streaks and count them only as part of streak of greater size. But let’s check slightly different example. In this case with such solution red cell will be never counted and it is wrong.

Slide 17.

Better solution is to detect and remember 1 on 1 streaks, when we are calculating number of white streaks in rows. But during calculation of white streaks in columns, we should skip all 1 on 1 streaks, if they’re part of streak of greater size in rows, or (which is the same statement), we can count them, only if it is already matched as the same 1 on 1 streak in rows.

For example, on the image red cell will be found as 1 on 1 streak in both parts: as column streak and as row streak. It means, that this streak isn’t a part of larger streak and should be calculated itself.

Slide 18.

Because if we find a 1 on 1 white streak during column calculations, we need to fast check, that there was the same 1 on 1 white streak during row calculations. The fastest way is to use set for this. During calculations of streaks in rows, such streak can be added to set. On the next step, during calculation of streaks in columns, it is easy to check set for presence of the same 1 on 1 streak. Search operation for set has logarithmic complexity.

So, the problem comes down to calculate sum of row streaks, column streaks and specifically handle 1 on 1 streaks.

Slide 19.

Mandatory task. You should implement source code for problem 1628 “White Streaks”, upload it to Timus system and pass all tests. After this you should prepare a report and send it via email. Please, use template document for your report and carefully set correct subject for the mail.

Slide 20.

You will have 1 problem for homework. It is problem 1650 “Billionaires”. You should solve it by yourself. Please, note, that report for this problem should contain explanation, which data structures were chosen.

Homework is optional, but successful completion of this problem can give you extra points for better grade.

Slide 21.

Let’s briefly check the tasks of the third test.

The first task was related to red-black trees.

Red-black tree are binary search tree. Also, it should follow a couple of rules.

- Every node is either red or black
- The root and every leaf is black
- If a node is red, then both its children are black
- For each node, all paths to descendant leaves contain the same number of black nodes

Slide 22.

Let's look at similar examples. For this tree is obvious, that tree isn’t binary search tree, because left child's value of root is greater, than value of root. Swapping of them is fixed the tree.

Slide 23.

Paths to all descendant leaves contain 2 black nodes, except leaves of node with value 12 – they have 3 black nodes in path. So, recoloring of all non-leaf nodes in right subtree of root will fix this tree.

Slide 24.

And this tree is correct because all requirements are satisfied.

Slide 25.

Task 2 was related to hash tables and collision resolution. Let's look at similar examples.

Before inserting any element, we have hash table, where all cells are empty.

Let’s insert element 4. Hash function gets value 4. So, add it to cell with index 4.

Slide 26.

Insert element 10. Hash function gets value 4. So, add it to cell with index 4. But cell with this index already has a value. Collision resolution method for this hash table is chaining. It means, that we can set to the cell multiple values (for example, cell can store a linked list of elements). Add 10 to this cell. Now it contains values 4 and 10.

Insert element 8. Hash function gets value 2. So, add it to cell with index 2

Slide 27.

Insert element 35. Hash function gets value 5. Add it to cell with index 5.

Insert element 7. Hash function gets value 1. Add it to cell with index 1.

Let’s check, which elements in the cell with index 4. It is elements 4 and 10.

Slide 28.

Another hash table. Before inserting any element, we have hash table, where all cells are empty.

Insert element 4. Hash function gets value 4. Add it to cell with index 4.

Slide 29.

Insert element 10. Hash function gets value 4. So, add it to cell with index 4. But cell with this index already has a value. Collision resolution method for this hash table is open addressing with linear probing. It means, that we should put this element to the next empty cell. Add this element to cell with index 5.

Insert element 8. Hash function gets value 2. Add it to cell with index 2.

Slide 30.

Insert element 35. Hash function gets value 5. Add it to cell with index 5. But cell with this index already has a value. Next empty cell has index 0.

Insert element 7. Hash function gets value 1. Add it to cell with index 1.

Let’s check, which elements in the cell with index 5. It is element 10.

Slide 31.

Another hash table. Before inserting any element, hash table is empty.

Insert element 2. Hash function gets value 2. So, add it to cell with index 2.

Slide 32.

Insert element 6. Hash function gets value 2. So, add it to cell with index 2. But cell with this index already has a value. Collision resolution method for this hash table is chaining. Now this cell has elements with value 2 and 6.

Insert element 18. Hash function gets value 2. So, add it to cell with index 2. Now this cell has elements with value 2, 6 , 18

Slide 33.

Insert element 23. Hash function gets value 3. Add it to cell with index 3.

Let’s check, which elements in the cell with index 0. There is no elements, so cell is empty.

Slide 34.

The third task is related to representation of graph with adjacency matrix. Let's look at similar examples.

Here we have directed graph without weights. Correct matrix on the slide. Value of cell should be equal to 1 for outgoing arrows and -1 for incoming arrows.

Slide 35.

It is undirected graph without weights, but it has loop for vertex two. Correct answer is on the slide. Matrix should be symmetric and has value 1 in position (2, 2)

Slide 36.

The last graph is undirected graph with weights. It is undirected, and there is no negative values – adjacency matrix shouldn’t have negative values. Weights should be correct in the matrix. Correct matrix on the slide.

Slide 37.

Answers to questions of task 4 can be found in lecture presentations

Short versions of answers:

Hash table iteration is slower than BST traversal

Main operations on balanced BST are guaranteed to run in O(log N), hash table operations are unbounded

Matrix requires $\Theta(n^2)$ memory, check edge between vertices – time is constant (O(1))

List: requires Θ(m) memory, check edge between vertices –is linear (O(n))

B-trees – self-balancing search trees which allow nodes to have more than one key

2-3-4 trees – a subset of B-trees where each node can have two, three or four children

Slide 38.

Thank you for attention