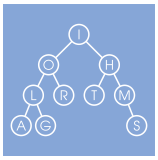


Module: Basic Data Structures (Week 1 out of 4)
Course: Data Structures (Course 2 out of 6)
Specialization: Data Structures and Algorithms

Programming Assignment 1: Basic Data Structures

Revision: December 8, 2016

Introduction

Welcome to your first Programming Assignment in the Data Structures course of the Data Structures and Algorithms Specialization!

In this programming assignment, you will be practicing implementing basic data structures and using them to solve algorithmic problems. In some of the problems, you just need to implement and use a data structure from the lectures, while in the others you will also need to invent an algorithm to solve the problem using some of the basic data structures.

In this programming assignment, the grader will show you the input and output data if your solution fails on any of the tests. This is done to help you to get used to the algorithmic problems in general and get some experience debugging your programs while knowing exactly on which tests they fail. However, note that for very big inputs the grader cannot show them fully, so it will only show the beginning of the input for you to make sense of its size, and then it will be clipped. You will need to generate big inputs for yourself. For all the following programming assignments, the grader will show the input data only in case your solution fails on one of the first few tests (please review the questions 5.4 and 5.5 in the FAQ section for a more detailed explanation of this behavior of the grader).

Learning Outcomes

Upon completing this programming assignment you will be able to:

1. Apply the basic data structures you've just studied to solve the given algorithmic problems.
2. Given a piece of code in an unknown programming language, check whether the brackets are used correctly in the code or not.
3. Implement a tree, read it from the input and compute its height.
4. Simulate processing of computer network packets.

Passing Criteria: 2 out of 3

Passing this programming assignment requires passing at least 2 out of 3 code problems from this assignment. In turn, passing a code problem requires implementing a solution that passes all the tests for this problem in the grader and does so under the time and memory limits specified in the problem statement.

Contents

1 Problem: Check brackets in the code

Problem Introduction

In this problem you will implement a feature for a text editor to find errors in the usage of brackets in the code.

Problem Description

Task. Your friend is making a text editor for programmers. He is currently working on a feature that will find errors in the usage of different types of brackets. Code can contain any brackets from the set `[]{}()`, where the opening brackets are `[`,`{`, and `(` and the closing brackets corresponding to them are `]`,`}`, and `)`.

For convenience, the text editor should not only inform the user that there is an error in the usage of brackets, but also point to the exact place in the code with the problematic bracket. First priority is to find the first unmatched closing bracket which either doesn't have an opening bracket before it, like `]` in `]()`, or closes the wrong opening bracket, like `}` in `()[}`. If there are no such mistakes, then it should find the first unmatched opening bracket without the corresponding closing bracket after it, like `(` in `{}([]`. If there are no mistakes, text editor should inform the user that the usage of brackets is correct.

Apart from the brackets, code can contain big and small latin letters, digits and punctuation marks.

More formally, all brackets in the code should be divided into pairs of matching brackets, such that in each pair the opening bracket goes before the closing bracket, and for any two pairs of brackets either one of them is nested inside another one as in `(foo[bar])` or they are separate as in `f(a,b)-g[c]`. The bracket `[` corresponds to the bracket `]`, `{` corresponds to `}`, and `(` corresponds to `)`.

Input Format. Input contains one string S which consists of big and small latin letters, digits, punctuation marks and brackets from the set `[]{}()`.

Constraints. The length of S is at least 1 and at most 10^5.

Output Format. If the code in S uses brackets correctly, output "Success" (without the quotes). Otherwise, output the 1-based index of the first unmatched closing bracket, and if there are no unmatched closing brackets, output the 1-based index of the first unmatched opening bracket.

Time Limits. `C`: 1 sec, `C++`: 1 sec, `Java`: 1 sec, `Python`: 1 sec. `C#`: 1.5 sec, `Haskell`: 2 sec, `JavaScript`: 3 sec, `Ruby`: 3 sec, `Scala`: 3 sec.

Memory Limit. 512MB.

Sample 1.

Input:

```
[]
```

Output:

```
Success
```

Explanation:

The brackets are used correctly: there is just one pair of brackets `[` and `]`, they correspond to each other, the left bracket `[` goes before the right bracket `]`, and no two pairs of brackets intersect, because there is just one pair of brackets.

Sample 2.

Input:

```
{}[]
```

Output:

```
Success
```

Explanation:
The brackets are used correctly: there are two pairs of brackets — first pair of `{` and `}`, and second pair of `[` and `]` — and these pairs do not intersect.

Sample 3.

Input:

```
[()]
```

Output:

```
Success
```

Explanation:
The brackets are used correctly: there are two pairs of brackets — first pair of `[` and `]`, and second pair of `(` and `)` — and the second pair is nested inside the first pair.

Sample 4.

Input:

```
(())
```

Output:

```
Success
```

Explanation:
Pairs with the same types of brackets can also be nested.

Sample 5.

Input:

```
{[]}()
```

Output:

```
Success
```

Explanation:
Here there are 3 pairs of brackets, one of them is nested into another one, and the third one is separate from the first two.

Sample 6.

Input:

```
{
```

Output:

```
1
```

Explanation:
The code `{` doesn't use brackets correctly, because brackets cannot be divided into pairs (there is just one bracket). There are no closing brackets, and the first unmatched opening bracket is `{`, and its position is 1, so we output 1.

Sample 7.

Input:

```
{[}
```

Output:

```
3
```

Explanation:
The bracket `}` is unmatched, because the last unmatched opening bracket before it is `[` and not `{`. It is the first unmatched closing bracket, and our first priority is to output the first unmatched closing bracket, and its position is 3, so we output 3.

Sample 8.

Input:

```
foo(bar);
```

Output:

```
Success
```

Explanation:
All the brackets are matching, and all the other symbols can be ignored.

Sample 9.

Input:

```
foo(bar[i);
```

Output:

```
10
```

Explanation:
`)` doesn't match `[`, so `)` is the first unmatched closing bracket, so we output its position, which is 10.

Starter Files

There are starter solutions only for C++, Java and Python3, and if you use other languages, you need to implement solution from scratch. Starter solutions read the code from the input and go through the code character-by-character and provide convenience methods. You need to implement the processing of the brackets to find the answer to the problem and to output the answer.

What to Do

To solve this problem, you can slightly modify the `IsBalanced` algorithm from the lectures to account not only for the brackets, but also for other characters in the code, and return not just whether the code uses brackets correctly, but also what is the first position where the code becomes broken.

Need Help?

Ask a question or see the questions asked by other learners at this forum thread.

2 Problem: Compute tree height

Problem Introduction

Trees are used to manipulate hierarchical data such as hierarchy of categories of a retailer or the directory structure on your computer. They are also used in data analysis and machine learning both for hierarchical clustering and building complex predictive models, including some of the best-performing in practice algorithms like Gradient Boosting over Decision Trees and Random Forests. In the later modules of this course, we will introduce balanced binary search trees (BST) — a special kind of trees that allows to very efficiently store, manipulate and retrieve data. Balanced BSTs are thus used in databases for efficient storage and actually in virtually any non-trivial programs, typically via built-in data structures of the programming language at hand.

In this problem, your goal is to get used to trees. You will need to read a description of a tree from the input, implement the tree data structure, store the tree and compute its height.

Problem Description

Task. You are given a description of a rooted tree. Your task is to compute and output its height. Recall that the height of a (rooted) tree is the maximum depth of a node, or the maximum distance from a leaf to the root. You are given an arbitrary tree, not necessarily a binary tree.

Input Format. The first line contains the number of nodes n. The second line contains n integer numbers from -1 to $n-1$ — parents of nodes. If the i-th one of them ($0 \le i \le n-1$) is -1, node i is the root, otherwise it's 0-based index of the parent of i-th node. It is guaranteed that there is exactly one root. It is guaranteed that the input represents a tree.

Constraints. $1 \le n \le 10^5$.

Output Format. Output the height of the tree.

Time Limits. `C`: 1 sec, `C++`: 1 sec, `Java`: 6 sec, `Python`: 3 sec. `C#`: 1.5 sec, `Haskell`: 2 sec, `JavaScript`: 3 sec, `Ruby`: 3 sec, `Scala`: 3 sec.

Memory Limit. 512MB.

Sample 1.

Input:

```
5
4 -1 4 1 1
```

Output:

```
3
```

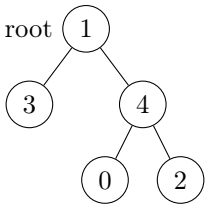
Explanation:

The input means that there are 5 nodes with numbers from 0 to 4, node 0 is a child of node 4, node 1 is the root, node 2 is a child of node 4, node 3 is a child of node 1 and node 4 is a child of node 1. To see this, let us write numbers of nodes from 0 to 4 in one line and the numbers given in the input in the second line underneath:

0 1 2 3 4

4 -1 4 1 1

Now we can see that the node number 1 is the root, because -1 corresponds to it in the second line. Also, we know that the nodes number 3 and number 4 are children of the root node 1. Also, we know that the nodes number 0 and number 2 are children of the node 4.



The height of this tree is 3, because the number of vertices on the path from root 1 to leaf 2 is 3.

Sample 2.

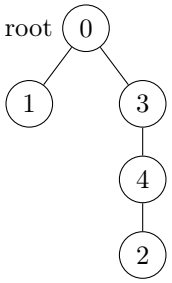
Input:

```
5
-1 0 4 0 3
```

Output:

```
4
```

Explanation:
The input means that there are 5 nodes with numbers from 0 to 4, node 0 is the root, node 1 is a child of node 0, node 2 is a child of node 4, node 3 is a child of node 0 and node 4 is a child of node 3. The height of this tree is 4, because the number of nodes on the path from root 0 to leaf 2 is 4.



Starter Files

The starter solutions in this problem read the description of a tree, store it in memory, compute the height in a naive way and write the output. You need to implement faster height computation. Starter solutions are available for C++, Java and Python3, and if you use other languages, you need to implement a solution from scratch.

What to Do

To solve this problem, change the height function described in the lectures with an implementation which will work for an arbitrary tree. Note that the tree can be very deep in this problem, so you should be careful to avoid stack overflow problems if you're using recursion, and definitely test your solution on a tree with the maximum possible height.

Suggestion: Take advantage of the fact that the labels for each tree node are integers in the range $0..n-1$: you can store each node in an array whose index is the label of the node. By storing the nodes in an array, you have $O(1)$ access to any node given its label.

Create an array of n nodes:

```
allocate nodes[n]
for i ← 0 to n − 1:
  nodes[i] =new Node
```

Then, read each parent index:

```
for child_index ← 0 to n − 1:
  read parent_index
  if parent_index == −1:
    root ← child_index
  else:
    nodes[parent_index].addChild(nodes[child_index])
```

Once you've built the tree, you'll then need to compute its height. If you don't use recursion, you needn't worry about stack overflow problems. Without recursion, you'll need some auxiliary data structure to keep track of the current state (in the breadth-first seach code in lecture, for example, we used a queue).

Need Help?

Ask a question or see the questions asked by other learners at this forum thread.

3 Advanced Problem: Network packet processing simulation

We strongly recommend you start solving advanced problems only when you are done with the basic problems (for some advanced problems, algorithms are not covered in the video lectures and require additional ideas to be solved; for some other advanced problems, algorithms are covered in the lectures, but implementing them is a more challenging task than for other problems).

Problem Introduction

In this problem you will implement a program to simulate the processing of network packets.

Problem Description

Task. You are given a series of incoming network packets, and your task is to simulate their processing. Packets arrive in some order. For each packet number i, you know the time when it arrived A_i and the time it takes the processor to process it P_i (both in milliseconds). There is only one processor, and it processes the incoming packets in the order of their arrival. If the processor started to process some packet, it doesn't interrupt or stop until it finishes the processing of this packet, and the processing of packet i takes exactly P_i milliseconds.

The computer processing the packets has a network buffer of fixed size S. When packets arrive, they are stored in the buffer before being processed. However, if the buffer is full when a packet arrives (there are S packets which have arrived before this packet, and the computer hasn't finished processing any of them), it is dropped and won't be processed at all. If several packets arrive at the same time, they are first all stored in the buffer (some of them may be dropped because of that — those which are described later in the input). The computer processes the packets in the order of their arrival, and it starts processing the next available packet from the buffer as soon as it finishes processing the previous one. If at some point the computer is not busy, and there are no packets in the buffer, the computer just waits for the next packet to arrive. Note that a packet leaves the buffer and frees the space in the buffer as soon as the computer finishes processing it.

Input Format. The first line of the input contains the size S of the buffer and the number n of incoming network packets. Each of the next n lines contains two numbers. i-th line contains the time of arrival A_i and the processing time P_i (both in milliseconds) of the i-th packet. It is guaranteed that the sequence of arrival times is non-decreasing (however, it can contain the exact same times of arrival in milliseconds — in this case the packet which is earlier in the input is considered to have arrived earlier).

Constraints. All the numbers in the input are integers. $1 \le S \le 10^5$; $1 \le n \le 10^5$; $0 \le A_i \le 10^6$; $0 \le P_i \le 10^3$; $A_i \le A_{i+1}$ for $1 \le i \le n-1$.

Output Format. For each packet output either the moment of time (in milliseconds) when the processor began processing it or -1 if the packet was dropped (output the answers for the packets in the same order as the packets are given in the input).

Time Limits. `C`: 2 sec, `C++`: 2 sec, `Java`: 6 sec, `Python`: 8 sec. `C#`: 3 sec, `Haskell`: 4 sec, `JavaScript`: 6 sec, `Ruby`: 6 sec, `Scala`: 6 sec.

Memory Limit. 512MB.

Sample 1.

Input:

```
1 0
```

Output:

Explanation:
If there are no packets, you shouldn't output anything.

Sample 2.

Input:

```
1 1
0 0
```

Output:

```
0
```

Explanation:
The only packet arrived at time 0, and computer started processing it immediately.

Sample 3.

Input:

```
1 2
0 1
0 1
```

Output:

```
0
-1
```

Explanation:
The first packet arrived at time 0, the second packet also arrived at time 0, but was dropped, because the network buffer has size 1 and it was full with the first packet already. The first packet started processing at time 0, and the second packet was not processed at all.

Sample 4.

Input:

```
1 2
0 1
1 1
```

Output:

```
0
1
```

Explanation:
The first packet arrived at time 0, the computer started processing it immediately and finished at time 1. The second packet arrived at time 1, and the computer started processing it immediately.

Starter Files

The starter solutions for C++, Java and Python3 in this problem read the input, pass the requests for processing of packets one-by-one and output the results. They declare a class that implements network buffer simulator. The class is partially implemented, and your task is to implement the rest of it. If you use other languages, you need to implement the solution from scratch.

What to Do

To solve this problem, you can use a list or a queue (in this case the queue should allow accessing its last element, and such queue is usually called a deque). You can use the corresponding built-in data structure in your language of choice.

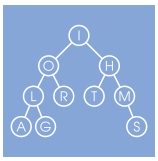
One possible solution is to store in the list or queue `finish_time` the times when the computer will finish processing the packets which are currently stored in the network buffer, in increasing order. When a new packet arrives, you will first need to pop from the front of `finish_time` all the packets which are already processed by the time new packet arrives. Then you try to add the finish time for the new packet in `finish_time`. If the buffer is full (there are already S finish times in `finish_time`), the packet is dropped. Otherwise, its processing finish time is added to `finish_time`.

If `finish_time` is empty when a new packet arrives, computer will start processing the new packet immediately as soon as it arrives. Otherwise, computer will start processing the new packet as soon as it finishes to process the last of the packets currently in `finish_time` (here is when you need to access the last element of `finish_time` to determine when the computer will start to process the new packet). You will also need to compute the processing finish time by adding P_i to the processing start time and push it to the back of `finish_time`.

You need to remember to output the processing start time for each packet instead of the processing finish time which you store in `finish_time`.

Need Help?

Ask a question or see the questions asked by other learners at this forum thread.



Module: Priority Queues and Disjoint Sets (Week 2 out of 4)
Course: Data Structures (Course 2 out of 6)
Specialization: Data Structures and Algorithms

Programming Assignment 2: Priority Queues and Disjoint Sets

Revision: August 26, 2016

Introduction

In this programming assignment, you will practice implementing priority queues and disjoint sets and using them to solve algorithmic problems. In some cases you will just implement an algorithm from the lectures, while in others you will need to invent an algorithm to solve the given problem using either a priority queue or a disjoint set union.

Recall that starting from this programming assignment, the grader will show you only the first few tests (see the questions 5.4 and 5.5 in the FAQ section).

Learning Outcomes

Upon completing this programming assignment you will be able to:

1. Apply priority queues and disjoint sets to solve the given algorithmic problems.
2. Convert an array into a heap.
3. Simulate a program which processes a list of jobs in parallel.
4. Simulate a sequence of merge operations with tables in a database.

Passing Criteria: 2 out of 3

Passing this programming assignment requires passing at least 2 out of 3 code problems from this assignment. In turn, passing a code problem requires implementing a solution that passes all the tests for this problem in the grader and does so under the time and memory limits specified in the problem statement.

Contents

1 Problem: Convert array into heap

Problem Introduction

In this problem you will convert an array of integers into a heap. This is the crucial step of the sorting algorithm called HeapSort. It has guaranteed worst-case running time of $O(n \log n)$ as opposed to QuickSort's average running time of $O(n \log n)$. QuickSort is usually used in practice, because typically it is faster, but HeapSort is used for external sort when you need to sort huge files that don't fit into memory of your computer.

Problem Description

Task. The first step of the HeapSort algorithm is to create a heap from the array you want to sort. By the way, did you know that algorithms based on Heaps are widely used for external sort, when you need to sort huge files that don't fit into memory of a computer?

Your task is to implement this first step and convert a given array of integers into a heap. You will do that by applying a certain number of swaps to the array. Swap is an operation which exchanges elements a_i and a_j of the array a for some i and j. You will need to convert the array into a heap using only $O(n)$ swaps, as was described in the lectures. Note that you will need to use a min-heap instead of a max-heap in this problem.

Input Format. The first line of the input contains single integer n. The next line contains n space-separated integers a_i.

Constraints. $1 \le n \le 100\,000$; $0 \le i, j \le n - 1$; $0 \le a_0, a_1, \ldots, a_{n-1} \le 10^9$. All a_i are distinct.

Output Format. The first line of the output should contain single integer m — the total number of swaps. m **must satisfy conditions** $0 \le m \le 4n$. The next m lines should contain the swap operations used to convert the array a into a heap. Each swap is described by a pair of integers i, j — the 0-based indices of the elements to be swapped. After applying all the swaps in the specified order the array must become a heap, that is, for each i where $0 \le i \le n - 1$ the following conditions must be true:

1. If $2i + 1 \le n - 1$, then $a_i < a_{2i+1}$.
2. If $2i + 2 \le n - 1$, then $a_i < a_{2i+2}$.

Note that all the elements of the input array are distinct. Note that any sequence of swaps that has length at most $4n$ and after which your initial array becomes a correct heap will be graded as correct.

Time Limits. `C`: 1 sec, `C++`: 1 sec, `Java`: 3 sec, `Python`: 3 sec. `C#`: 1.5 sec, `Haskell`: 2 sec, `JavaScript`: 3 sec, `Ruby`: 3 sec, `Scala`: 3 sec.

Memory Limit. 512Mb.

Sample 1.

Input:

```
5
5 4 3 2 1
```

Output:

```
3
1 4
0 1
1 3
```

Explanation:
After swapping elements 4 in position 1 and 1 in position 4 the array becomes 5 1 3 2 4.

After swapping elements 5 in position 0 and 1 in position 1 the array becomes 1 5 3 2 4.
After swapping elements 5 in position 1 and 2 in position 3 the array becomes 1 2 3 5 4, which is already a heap, because $a_0 = 1 < 2 = a_1, a_0 = 1 < 3 = a_2, a_1 = 2 < 5 = a_3, a_1 = 2 < 4 = a_4$.

Sample 2.

Input:

```
5
1 2 3 4 5
```

Output:

```
0
```

Explanation:
The input array is already a heap, because it is sorted in increasing order.

Starter Files

There are starter solutions only for C++, Java and Python3, and if you use other languages, you need to implement solution from scratch. Starter solutions read the array from the input, use a quadratic time algorithm to convert it to a heap and use $\Theta(n^2)$ swaps to do that, then write the output. You need to replace the $\Theta(n^2)$ implementation with an $O(n)$ implementation using no more than $4n$ swaps to convert the array into heap.

What to Do

Change the `BuildHeap` algorithm from the lecture to account for min-heap instead of max-heap and for 0-based indexing.

Need Help?

Ask a question or see the questions asked by other learners at this forum thread.

2 Problem: Parallel processing

Problem Introduction

In this problem you will simulate a program that processes a list of jobs in parallel. Operating systems such as Linux, MacOS or Windows all have special programs in them called schedulers which do exactly this with the programs on your computer.

Problem Description

Task. You have a program which is parallelized and uses n independent threads to process the given list of m jobs. Threads take jobs in the order they are given in the input. If there is a free thread, it immediately takes the next job from the list. If a thread has started processing a job, it doesn't interrupt or stop until it finishes processing the job. If several threads try to take jobs from the list simultaneously, the thread with smaller index takes the job. For each job you know exactly how long will it take any thread to process this job, and this time is the same for all the threads. You need to determine for each job which thread will process it and when will it start processing.

Input Format. The first line of the input contains integers n and m.
The second line contains m integers t_i — the times in seconds it takes any thread to process i-th job. The times are given in the same order as they are in the list from which threads take jobs.
Threads are indexed starting from 0.

Constraints. $1 \le n \le 10^5$; $1 \le m \le 10^5$; $0 \le t_i \le 10^9$.

Output Format. Output exactly m lines. i-th line (0-based index is used) should contain two space-separated integers — the 0-based index of the thread which will process the i-th job and the time in seconds when it will start processing that job.

Time Limits. `C`: 1 sec, `C++`: 1 sec, `Java`: 4 sec, `Python`: 6 sec. `C#`: 1.5 sec, `Haskell`: 2 sec, `JavaScript`: 6 sec, `Ruby`: 6 sec, `Scala`: 6 sec.

Memory Limit. 512Mb.

Sample 1.

Input:

```
2 5
1 2 3 4 5
```

Output:

```
0 0
1 0
0 1
1 2
0 4
```

Explanation:

1. The two threads try to simultaneously take jobs from the list, so thread with index 0 actually takes the first job and starts working on it at the moment 0.
2. The thread with index 1 takes the second job and starts working on it also at the moment 0.
3. After 1 second, thread 0 is done with the first job and takes the third job from the list, and starts processing it immediately at time 1.
4. One second later, thread 1 is done with the second job and takes the fourth job from the list, and starts processing it immediately at time 2.

5. Finally, after 2 more seconds, thread 0 is done with the third job and takes the fifth job from the list, and starts processing it immediately at time 4.

Sample 2.

Input:

```
4 20
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Output:

```
0 0
1 0
2 0
3 0
0 1
1 1
2 1
3 1
0 2
1 2
2 2
3 2
0 3
1 3
2 3
3 3
0 4
1 4
2 4
3 4
```

Explanation:
Jobs are taken by 4 threads in packs of 4, processed in 1 second, and then the next pack comes. This happens 5 times starting at moments 0, 1, 2, 3 and 4. After that all the $5 \times 4 = 20$ jobs are processed.

Starter Files

The starter solutions for C++, Java and Python3 in this problem read the input, apply an $\Theta(n^2)$ algorithm to solve the problem and write the output. You need to replace the $\Theta(n^2)$ algorithm with a faster one. If you use other languages, you need to implement the solution from scratch.

What to Do

Think about the sequence of events when one of the threads becomes free (at the start and later after completing some job). How to apply priority queue to simulate processing of these events in the required order? Remember to consider the case when several threads become free simultaneously.

Beware of integer overflow in this problem: use type `long long` in C++ and type `long` in Java wherever the regular type `int` can overflow given the restrictions in the problem statement.

Need Help?

Ask a question or see the questions asked by other learners at this forum thread.

3 Problem: Merging tables

Problem Introduction

In this problem, your goal is to simulate a sequence of merge operations with tables in a database.

Problem Description

Task. There are n tables stored in some database. The tables are numbered from 1 to n. All tables share the same set of columns. Each table contains either several rows with real data or a symbolic link to another table. Initially, all tables contain data, and i-th table has r_i rows. You need to perform m of the following operations:

1. Consider table number $destination_i$. Traverse the path of symbolic links to get to the data. That is,

   while $destination_i$ contains a symbolic link instead of real data do

   $destination_i \leftarrow \texttt{symlink}(destination_i)$

2. Consider the table number $source_i$ and traverse the path of symbolic links from it in the same manner as for $destination_i$.
3. Now, $destination_i$ and $source_i$ are the numbers of two tables with real data. If $destination_i \neq source_i$, copy all the rows from table $source_i$ to table $destination_i$, then clear the table $source_i$ and instead of real data put a symbolic link to $destination_i$ into it.
4. Print the maximum size among all n tables (recall that size is the number of rows in the table). If the table contains only a symbolic link, its size is considered to be 0.

See examples and explanations for further clarifications.

Input Format. The first line of the input contains two integers n and m — the number of tables in the database and the number of merge queries to perform, respectively.
The second line of the input contains n integers r_i — the number of rows in the i-th table.
Then follow m lines describing merge queries. Each of them contains two integers $destination_i$ and $source_i$ — the numbers of the tables to merge.

Constraints. $1 \leq n, m \leq 100\,000$; $0 \leq r_i \leq 10\,000$; $1 \leq destination_i, source_i \leq n$.

Output Format. For each query print a line containing a single integer — the maximum of the sizes of all tables (in terms of the number of rows) after the corresponding operation.

Time Limits. `C`: 2 sec, `C++`: 2 sec, `Java`: 14 sec, `Python`: 6 sec. `C#`: 3 sec, `Haskell`: 4 sec, `JavaScript`: 6 sec, `Ruby`: 6 sec, `Scala`: 14 sec.

Memory Limit. 512Mb.

Sample 1.

Input:

```
5 5
1 1 1 1 1
3 5
2 4
1 4
5 4
5 3
```

Output:

```
2
2
3
5
5
```

Explanation:
In this sample, all the tables initially have exactly 1 row of data. Consider the merging operations:

1. All the data from the table 5 is copied to table number 3. Table 5 now contains only a symbolic link to table 3, while table 3 has 2 rows. 2 becomes the new maximum size.
2. 2 and 4 are merged in the same way as 3 and 5.
3. We are trying to merge 1 and 4, but 4 has a symbolic link pointing to 2, so we actually copy all the data from the table number 2 to the table number 1, clear the table number 2 and put a symbolic link to the table number 1 in it. Table 1 now has 3 rows of data, and 3 becomes the new maximum size.
4. Traversing the path of symbolic links from 4 we have $4 \to 2 \to 1$, and the path from 5 is $5 \to 3$. So we are actually merging tables 3 and 1. We copy all the rows from the table number 1 into the table number 3, and now the table number 3 has 5 rows of data, which is the new maximum.
5. All tables now directly or indirectly point to table 3, so all other merges won't change anything.

Sample 2.

Input:

```
6 4
10 0 5 0 3 3
6 6
6 5
5 4
4 3
```

Output:

```
10
10
10
11
```

Explanation:
In this example tables have different sizes. Let us consider the operations:

1. Merging the table number 6 with itself doesn't change anything, and the maximum size is 10 (table number 1).

2. After merging the table number 5 into the table number 6, the table number 5 is cleared and has size 0, while the table number 6 has size 6. Still, the maximum size is 10.
3. By merging the table number 4 into the table number 5, we actually merge the table number 4 into the table number 6 (table 5 now contains just a symbolic link to table 6), so the table number 4 is cleared and has size 0, while the table number 6 has size 6. Still, the maximum size is 10.
4. By merging the table number 3 into the table number 4, we actually merge the table number 3 into the table number 6 (table 4 now contains just a symbolic link to table 6), so the table number 3 is cleared and has size 0, while the table number 6 has size 11, which is the new maximum size.

Starter Files

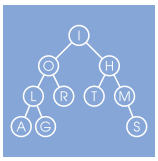
The starter solutions in C++, Java and Python3 read the description of tables and operations from the input, declare and partially implement disjoint set union, and write the output. You need to complete the implementation of disjoint set union for this problem. If you use other languages, you will have to implement the solution from scratch.

What to Do

Think how to use disjoint set union with path compression and union by rank heuristics to solve this problem. In particular, you should separate in your thinking the data structure that performs union/find operations from the merges of tables. If you're asked to merge first table into second, but the rank of the second table is smaller than the rank of the first table, you can ignore the requested order while merging in the Disjoint Set Union data structure and join the node corresponding to the second table to the node corresponding to the first table instead in you Disjoint Set Union. However, you will need to store the number of the actual second table to which you were requested to merge the first table in the parent node of the corresponding Disjoint Set, and you will need an additional field in the nodes of Disjoint Set Union to store it.

Need Help?

Ask a question or see the questions asked by other learners at this forum thread.



Module: Hash Tables (Week 3 out of 4)
Course: Data Structures (Course 2 out of 6)
Specialization: Data Structures and Algorithms

Programming Assignment 3: Hash Tables and Hash Functions

Revision: August 25, 2016

Introduction

In this programming assignment, you will practice implementing hash functions and hash tables and using them to solve algorithmic problems. In some cases you will just implement an algorithm from the lectures, while in others you will need to invent an algorithm to solve the given problem using hashing.

Learning Outcomes

Upon completing this programming assignment you will be able to:

1. Apply hashing to solve the given algorithmic problems.
2. Implement a simple phone book manager.
3. Implement a hash table using the chaining scheme.
4. Find all occurrences of a pattern in text using Rabin–Karp's algorithm.

Passing Criteria: 2 out of 3

Passing this programming assignment requires passing at least 2 out of 3 code problems from this assignment. In turn, passing a code problem requires implementing a solution that passes all the tests for this problem in the grader and does so under the time and memory limits specified in the problem statement.

Contents

1 Problem: Phone book

Problem Introduction

In this problem you will implement a simple phone book manager.

Problem Description

Task. In this task your goal is to implement a simple phone book manager. It should be able to process the following types of user's queries:

- **add** `number` `name`. It means that the user adds a person with name `name` and phone number `number` to the phone book. If there exists a user with such number already, then your manager has to overwrite the corresponding name.
- **del** `number`. It means that the manager should erase a person with number `number` from the phone book. If there is no such person, then it should just ignore the query.
- **find** `number`. It means that the user looks for a person with phone number `number`. The manager should reply with the appropriate name, or with string "not found" (without quotes) if there is no such person in the book.

Input Format. There is a single integer N in the first line — the number of queries. It's followed by N lines, each of them contains one query in the format described above.

Constraints. $1 \le N \le 10^5$. All phone numbers consist of decimal digits, they don't have leading zeros, and each of them has no more than 7 digits. All names are non-empty strings of latin letters, and each of them has length at most 15. It's guaranteed that there is no person with name "not found".

Output Format. Print the result of each **find** query — the name corresponding to the phone number or "not found" (without quotes) if there is no person in the phone book with such phone number. Output one result per line in the same order as the **find** queries are given in the input.

Time Limits. `C`: 3 sec, `C++`: 3 sec, `Java`: 6 sec, `Python`: 6 sec. `C#`: 4.5 sec, `Haskell`: 6 sec, `JavaScript`: 9 sec, `Ruby`: 9 sec, `Scala`: 9 sec.

Memory Limit. 512Mb.

Sample 1.

Input:

```
12
add 911 police
add 76213 Mom
add 17239 Bob
find 76213
find 910
find 911
del 910
del 911
find 911
find 76213
add 76213 daddy
find 76213
```

Output:

```
Mom
not found
police
not found
Mom
daddy
```

Explanation:
76213 is Mom's number, 910 is not a number in the phone book, 911 is the number of police, but then it was deleted from the phone book, so the second search for 911 returned "not found". Also, note that when the daddy was added with the same phone number 76213 as Mom's phone number, the contact's name was rewritten, and now search for 76213 returns "daddy" instead of "Mom".

Sample 2.

Input:

```
8
find 3839442
add 123456 me
add 0 granny
find 0
find 123456
del 0
del 0
find 0
```

Output:

```
not found
granny
me
not found
```

Explanation:
Recall that deleting a number that doesn't exist in the phone book doesn't change anything.

Starter Files

The starter solutions for C++, Java and Python3 in this problem read the input, implement a naive algorithm to look up names by phone numbers and write the output. You need to use a fast data structure to implement a better algorithm. If you use other languages, you need to implement the solution from scratch.

What to Do

Use the direct addressing scheme.

Need Help?

Ask a question or see the questions asked by other learners at this forum thread.

2 Problem: Hashing with chains

Problem Introduction

In this problem you will implement a hash table using the chaining scheme. Chaining is one of the most popular ways of implementing hash tables in practice. The hash table you will implement can be used to implement a phone book on your phone or to store the password table of your computer or web service (but don't forget to store hashes of passwords instead of the passwords themselves, or you will get hacked!).

Problem Description

Task. In this task your goal is to implement a hash table with lists chaining. You are already given the number of buckets m and the hash function. It is a polynomial hash function

$$h(S) = \left(\sum_{i=0}^{|S|-1} S[i]x^i \bmod p \right) \bmod m\,,$$

where $S[i]$ is the ASCII code of the i-th symbol of S, $p = 1\ 000\ 000\ 007$ and $x = 263$. Your program should support the following kinds of queries:

- **add** `string` — insert `string` into the table. If there is already such string in the hash table, then just ignore the query.
- **del** `string` — remove `string` from the table. If there is no such string in the hash table, then just ignore the query.
- **find** `string` — output "yes" or "no" (without quotes) depending on whether the table contains `string` or not.
- **check** i — output the content of the i-th list in the table. Use spaces to separate the elements of the list. **If i-th list is empty, output a blank line.**

When inserting a new string into a hash chain, you must insert it in the beginning of the chain.

Input Format. There is a single integer m in the first line — the number of buckets you should have. The next line contains the number of queries N. It's followed by N lines, each of them contains one query in the format described above.

Constraints. $1 \le N \le 10^5$; $\frac{N}{5} \le m \le N$. All the strings consist of latin letters. Each of them is non-empty and has length at most 15.

Output Format. Print the result of each of the **find** and **check** queries, one result per line, in the same order as these queries are given in the input.

Time Limits. `C`: 1 sec, `C++`: 1 sec, `Java`: 5 sec, `Python`: 7 sec. `C#`: 1.5 sec, `Haskell`: 2 sec, `JavaScript`: 7 sec, `Ruby`: 7 sec, `Scala`: 7 sec.

Memory Limit. 512Mb.

Sample 1.

Input:

```
5
12
add world
add HellO
check 4
find World
find world
del world
check 4
del HellO
add luck
add GooD
check 2
del good
```

Output:

```
HellO world
no
yes
HellO
GooD luck
```

Explanation:
The ASCII code of 'w' is 119, for 'o' it is 111, for 'r' it is 114, for 'l' it is 108, and for 'd' it is 100. Thus, $h(\text{“world"}) = (119 + 111 \times 263 + 114 \times 263^2 + 108 \times 263^3 + 100 \times 263^4 \bmod 1\,000\,000\,007) \bmod 5 = 4$. It turns out that the hash value of *HellO* is also 4. Recall that we always insert in the beginning of the chain, so after adding “world" and then “HellO" in the same chain index 4, first goes “HellO" and then goes “world". Of course, “World" is not found, and “world" is found, because the strings are case-sensitive, and the codes of 'W' and 'w' are different. After deleting “world", only “HellO" is found in the chain 4. Similarly to “world" and “HellO", after adding “luck" and “GooD" to the same chain 2, first goes “GooD" and then “luck".

Sample 2.

Input:

```
4
8
add test
add test
find test
del test
find test
find Test
add Test
find Test
```

Output:

```
yes
no
no
yes
```

Explanation:
Adding “test" twice is the same as adding “test" once, so first **find** returns “yes". After del, “test" is

no longer in the hash table. First time **find** doesn't find "Test" because it was not added before, and strings are case-sensitive in this problem. Second time "Test" can be found, because it has just been added.

Sample 3.

Input:

```
3
12
check 0
find help
add help
add del
add add
find add
find del
del del
find del
check 0
check 1
check 2
```

Output:

```
no
yes
yes
no

add help
```

Explanation:
Note that you need to output a blank line when you handle an empty chain. Note that the strings stored in the hash table can coincide with the commands used to work with the hash table.

Starter Files

There are starter solutions only for C++, Java and Python3, and if you use other languages, you need to implement solution from scratch. Starter solutions read the input, do a full scan of the whole table to simulate each **find** operation and write the output. This naive simulation algorithm is too slow, so you need to implement the real hash table.

What to Do

Follow the explanations about the chaining scheme from the lectures. Remember to always insert new strings in the beginning of the chain. Remember to output a blank line when **check** operation is called on an empty chain.

Some hints based on the problems encountered by learners:

- Beware of integer overflow. Use `long long` type in C++ and `long` type in Java where appropriate. Take everything $\pmod p$ as soon as possible while computing something $\pmod p$, so that the numbers are always between 0 and $p - 1$.

- Beware of taking negative numbers $\pmod{p}$. In many programming languages, $(-2)\%5 \neq 3\%5$. Thus you can compute the same hash values for two strings, but when you compare them, they appear to be different. To avoid this issue, you can use such construct in the code: $x \leftarrow ((a\%p) + p)\%p$ instead of just $x \leftarrow a\%p$.

Need Help?

Ask a question or see the questions asked by other learners at this forum thread.

3 Problem: Find pattern in text

Problem Introduction

In this problem, your goal is to implement the Rabin–Karp's algorithm.

Problem Description

Task. In this problem your goal is to implement the Rabin–Karp's algorithm for searching the given pattern in the given text.

Input Format. There are two strings in the input: the pattern P and the text T.

Constraints. $1 \le |P| \le |T| \le 5 \cdot 10^5$. The total length of all occurrences of P in T doesn't exceed 10^8. The pattern and the text contain only latin letters.

Output Format. Print all the positions of the occurrences of P in T in the ascending order. Use 0-based indexing of positions in the the text T.

Time Limits. `C`: 1 sec, `C++`: 1 sec, `Java`: 5 sec, `Python`: 5 sec. `C#`: 1.5 sec, `Haskell`: 2 sec, `JavaScript`: 3 sec, `Ruby`: 3 sec, `Scala`: 3 sec.

Memory Limit. 512Mb.

Sample 1.

Input:

```
aba
abacaba
```

Output:

```
0 4
```

Explanation:
The pattern *aba* can be found in positions 0 (**aba**caba) and 4 (abac**aba**) of the text *abacaba*.

Sample 2.

Input:

```
Test
testTesttesT
```

Output:

```
4
```

Explanation:
Pattern and text are case-sensitive in this problem. Pattern $Test$ can only be found in position 4 in the text $testTesttesT$.

Sample 3.

Input:

```
aaaaa
baaaaaaa
```

Output:

```
1 2 3
```

Explanation:
Note that the occurrences of the pattern in the text can be overlapping, and that's ok, you still need to output all of them.

Starter Files

The starter solutions in C++, Java and Python3 read the input, apply the naive $O(|T||P|)$ algorithm to this problem and write the output. You need to implement the Rabin–Karp's algorithm instead of the naive algorithm and thus significantly speed up the solution. If you use other languages, you need to implement a solution from scratch.

What to Do

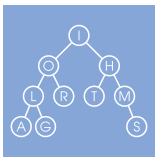
Implement the fast version of the Rabin–Karp's algorithm from the lectures.

Some hints based on the problems encountered by learners:

- Beware of integer overflow. Use `long long` type in C++ and `long` type in Java where appropriate. Take everything $\pmod p$ as soon as possible while computing something $\pmod p$, so that the numbers are always between 0 and $p - 1$.
- Beware of taking negative numbers $\pmod p$. In many programming languages, $(-2)\%5 \neq 3\%5$. Thus you can compute the same hash values for two strings, but when you compare them, they appear to be different. To avoid this issue, you can use such construct in the code: $x \leftarrow ((a\%p) + p)\%p$ instead of just $x \leftarrow a\%p$.
- Use operator `==` in Python instead of implementing your own function `AreEqual` for strings, because built-in operator `==` will work much faster.
- In C++, method `substr` of `string` creates a new string, uses additional memory and time for that, so use it carefully and avoid creating lots of new strings. When you need to compare pattern with a substring of text, do it without calling `substr`.
- In Java, however, method `substring` does NOT create a new `String`. Avoid using `new String` where it is not needed, just use `substring`.

Need Help?

Ask a question or see the questions asked by other learners at this forum thread.



Module: Binary Search Trees (Week 4 out of 4)
Course: Data Structures (Course 2 out of 6)
Specialization: Data Structures and Algorithms

Programming Assignment 4: Binary Search Trees

Revision: March 4, 2017

Introduction

In this programming assignment, you will practice implementing binary search trees including balanced ones and using them to solve algorithmic problems. In some cases you will just implement an algorithm from the lectures, while in others you will need to invent an algorithm to solve the given problem using hashing.

Learning Outcomes

Upon completing this programming assignment you will be able to:

1. Apply binary search trees to solve the given algorithmic problems.
2. Implement in-order, pre-order and post-order traversal of a binary tree.
3. Implement a data structure to compute range sums.
4. Implement a data structure that can store strings and quickly cut parts and patch them back.

Passing Criteria: 3 out of 5

Passing this programming assignment requires passing at least 3 out of 5 code problems from this assignment. In turn, passing a code problem requires implementing a solution that passes all the tests for this problem in the grader and does so under the time and memory limits specified in the problem statement.

Contents

1 Problem: Binary tree traversals

Problem Introduction

In this problem you will implement in-order, pre-order and post-order traversals of a binary tree. These traversals were defined in the week 1 lecture on tree traversals, but it is very useful to practice implementing them to understand binary search trees better.

Problem Description

Task. You are given a rooted binary tree. Build and output its in-order, pre-order and post-order traversals.

Input Format. The first line contains the number of vertices n. The vertices of the tree are numbered from 0 to $n - 1$. Vertex 0 is the root.

The next n lines contain information about vertices 0, 1, ..., $n-1$ in order. Each of these lines contains three integers key_i, $left_i$ and $right_i$ — key_i is the key of the i-th vertex, $left_i$ is the index of the left child of the i-th vertex, and $right_i$ is the index of the right child of the i-th vertex. If i doesn't have left or right child (or both), the corresponding $left_i$ or $right_i$ (or both) will be equal to -1.

Constraints. $1 \le n \le 10^5$; $0 \le key_i \le 10^9$; $-1 \le left_i, right_i \le n - 1$. It is guaranteed that the input represents a valid binary tree. In particular, if $left_i \ne -1$ and $right_i \ne -1$, then $left_i \ne right_i$. Also, a vertex cannot be a child of two different vertices. Also, each vertex is a descendant of the root vertex.

Output Format. Print three lines. The first line should contain the keys of the vertices in the in-order traversal of the tree. The second line should contain the keys of the vertices in the pre-order traversal of the tree. The third line should contain the keys of the vertices in the post-order traversal of the tree.

Time Limits.

language	C	C++	Java	Python	C#	Haskell	JavaScript	Ruby	Scala
time (sec)	1	1	12	6	1.5	2	6	6	12

Memory Limit. 512MB.

Sample 1.

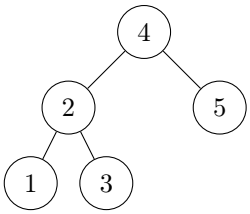
Input:

```
5
4 1 2
2 3 4
5 -1 -1
1 -1 -1
3 -1 -1
```

Output:

```
1 2 3 4 5
4 2 1 3 5
1 3 2 5 4
```

Explanation:



Sample 2.

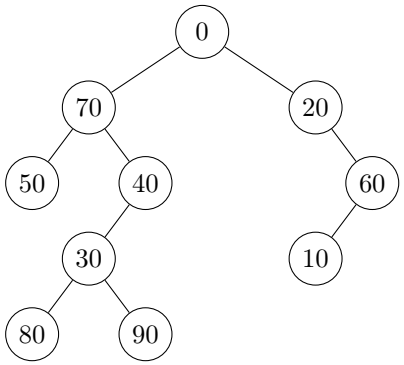
Input:

```
10
0 7 2
10 -1 -1
20 -1 6
30 8 9
40 3 -1
50 -1 -1
60 1 -1
70 5 4
80 -1 -1
90 -1 -1
```

Output:

```
50 70 80 30 90 40 0 20 10 60
0 70 50 40 30 80 90 20 60 10
50 80 90 30 40 70 10 60 20 0
```

Explanation:



Starter Files

There are starter solutions only for C++, Java and Python3, and if you use other languages, you need to implement solution from scratch. Starter solutions read the input, define the methods to compute different traversals of the binary tree and write the output. You need to implement the traversal methods.

What to Do

Implement the traversal algorithms from the lectures. Note that the tree can be very deep in this problem, so you should be careful to avoid stack overflow problems if you're using recursion, and definitely test your solution on a tree with the maximum possible height.

Need Help?

Ask a question or see the questions asked by other learners at this forum thread.

2 Problem: Is it a binary search tree?

Problem Introduction

In this problem you are going to test whether a binary search tree data structure from some programming language library was implemented correctly. There is already a program that plays with this data structure by inserting, removing, searching integers in the data structure and outputs the state of the internal binary tree after each operation. Now you need to test whether the given binary tree is indeed a correct binary search tree. In other words, you want to ensure that you can search for integers in this binary tree using binary search through the tree, and you will always get correct result: if the integer is in the tree, you will find it, otherwise you will not.

Problem Description

Task. You are given a binary tree with integers as its keys. You need to test whether it is a correct binary search tree. The definition of the binary search tree is the following: for any node of the tree, if its key is x, then for any node in its left subtree its key must be strictly less than x, and for any node in its right subtree its key must be strictly greater than x. In other words, smaller elements are to the left, and bigger elements are to the right. You need to check whether the given binary tree structure satisfies this condition. You are guaranteed that the input contains a valid binary tree. That is, it is a tree, and each node has at most two children.

Input Format. The first line contains the number of vertices n. The vertices of the tree are numbered from 0 to $n - 1$. Vertex 0 is the root.

The next n lines contain information about vertices 0, 1, ..., $n-1$ in order. Each of these lines contains three integers key_i, $left_i$ and $right_i$ — key_i is the key of the i-th vertex, $left_i$ is the index of the left child of the i-th vertex, and $right_i$ is the index of the right child of the i-th vertex. If i doesn't have left or right child (or both), the corresponding $left_i$ or $right_i$ (or both) will be equal to -1.

Constraints. $0 \le n \le 10^5$; $-2^{31} < key_i < 2^{31} - 1$; $-1 \le left_i, right_i \le n - 1$. It is guaranteed that the input represents a valid binary tree. In particular, if $left_i \ne -1$ and $right_i \ne -1$, then $left_i \ne right_i$. Also, a vertex cannot be a child of two different vertices. Also, each vertex is a descendant of the root vertex. All keys in the input will be different.

Output Format. If the given binary tree is a correct binary search tree (see the definition in the problem description), output one word "CORRECT" (without quotes). Otherwise, output one word "INCORRECT" (without quotes).

Time Limits.

language	`C`	`C++`	`Java`	`Python`	`C#`	`Haskell`	`JavaScript`	`Ruby`	`Scala`
time (sec)	2	2	3	10	3	4	10	10	6

Memory Limit. 512MB.

Sample 1.

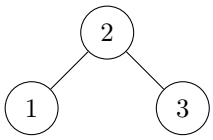
Input:

```
3
2 1 2
1 -1 -1
3 -1 -1
```

Output:

```
CORRECT
```

Explanation:



Left child of the root has key 1, right child of the root has key 3, root has key 2, so everything to the left is smaller, everything to the right is bigger.

Sample 2.

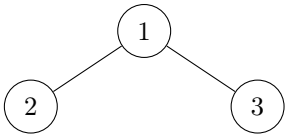
Input:

```
3
1 1 2
2 -1 -1
3 -1 -1
```

Output:

```
INCORRECT
```

Explanation:



The left child of the root must have smaller key than the root.

Sample 3.

Input:

```
0
```

Output:

```
CORRECT
```

Explanation:
Empty tree is considered correct.

Sample 4.

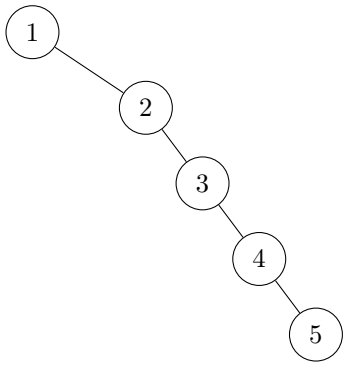
Input:

```
5
1 -1 1
2 -1 2
3 -1 3
4 -1 4
5 -1 -1
```

Output:

```
CORRECT
```

Explanation:



The tree doesn't have to be balanced. We only need to test whether it is a correct binary search tree, which the tree in this example is.

Sample 5.

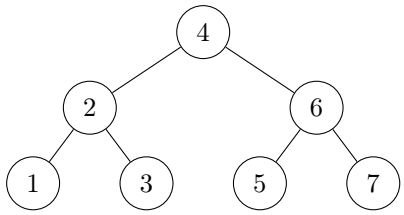
Input:

```
7
4 1 2
2 3 4
6 5 6
1 -1 -1
3 -1 -1
5 -1 -1
7 -1 -1
```

Output:

```
CORRECT
```

Explanation:



This is a full binary tree, and the property of the binary search tree is satisfied in every node.

Sample 6.

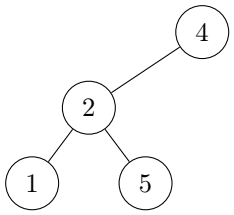
Input:

```
4
4 1 -1
2 2 3
1 -1 -1
5 -1 -1
```

Output:

```
INCORRECT
```

Explanation:



Node 5 is in the left subtree of the root, but it is bigger than the key 4 in the root.

Starter Files

The starter solutions for this problem read the input data from the standard input, pass it to a blank procedure, and then write the result to the standard output. You are supposed to implement your algorithm in this blank procedure if you are using `C++`, `Java`, or `Python3`. For other programming languages, you need to implement a solution from scratch. Filename: `is_bst`

What to Do

Testing the binary search tree condition for each node and every other node in its subtree will be too slow. You should come up with a faster algorithm.

Need Help?

Ask a question or see the questions asked by other learners at this forum thread.

3 Problem: Is it a binary search tree? Hard version.

Problem Introduction

In this problem you are going to solve the same problem as the previous one, but for a more general case, when binary search tree may contain equal keys.

Problem Description

Task. You are given a binary tree with integers as its keys. You need to test whether it is a correct binary search tree. Note that there can be duplicate integers in the tree, and this is allowed. The definition of the binary search tree in such case is the following: for any node of the tree, if its key is x, then for any node in its left subtree its key must be strictly less than x, and for any node in its right subtree its key must be greater than **or equal** to x. In other words, smaller elements are to the left, bigger elements are to the right, and duplicates are always to the right. You need to check whether the given binary tree structure satisfies this condition. You are guaranteed that the input contains a valid binary tree. That is, it is a tree, and each node has at most two children.

Input Format. The first line contains the number of vertices n. The vertices of the tree are numbered from 0 to $n - 1$. Vertex 0 is the root.

The next n lines contain information about vertices 0, 1, ..., $n-1$ in order. Each of these lines contains three integers key_i, $left_i$ and $right_i$ — key_i is the key of the i-th vertex, $left_i$ is the index of the left child of the i-th vertex, and $right_i$ is the index of the right child of the i-th vertex. If i doesn't have left or right child (or both), the corresponding $left_i$ or $right_i$ (or both) will be equal to -1.

Constraints. $0 \le n \le 10^5$; $-2^{31} \le key_i \le 2^{31} - 1$; $-1 \le left_i, right_i \le n - 1$. It is guaranteed that the input represents a valid binary tree. In particular, if $left_i \ne -1$ and $right_i \ne -1$, then $left_i \ne right_i$. Also, a vertex cannot be a child of two different vertices. Also, each vertex is a descendant of the root vertex. Note that the minimum and the maximum possible values of the 32-bit integer type are allowed to be keys in the tree — beware of integer overflow!

Output Format. If the given binary tree is a correct binary search tree (see the definition in the problem description), output one word "CORRECT" (without quotes). Otherwise, output one word "INCORRECT" (without quotes).

Time Limits.

language	`C`	`C++`	`Java`	`Python`	`C#`	`Haskell`	`JavaScript`	`Ruby`	`Scala`
time (sec)	2	2	3	10	3	4	10	10	6

Memory Limit. 512MB.

Sample 1.

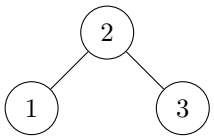
Input:

```
3
2 1 2
1 -1 -1
3 -1 -1
```

Output:

```
CORRECT
```

Explanation:



Left child of the root has key 1, right child of the root has key 3, root has key 2, so everything to the left is smaller, everything to the right is bigger.

Sample 2.

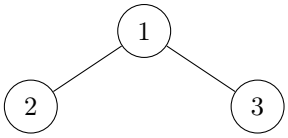
Input:

```
3
1 1 2
2 -1 -1
3 -1 -1
```

Output:

```
INCORRECT
```

Explanation:



The left child of the root must have smaller key than the root.

Sample 3.

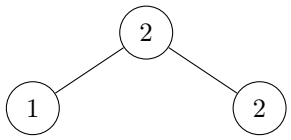
Input:

```
3
2 1 2
1 -1 -1
2 -1 -1
```

Output:

```
CORRECT
```

Explanation:



Duplicate keys are allowed, and they should always be in the right subtree of the first duplicated element.

Sample 4.

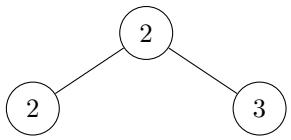
Input:

```
3
2 1 2
2 -1 -1
3 -1 -1
```

Output:

```
INCORRECT
```

Explanation:



The key of the left child of the root must be strictly smaller than the key of the root.

Sample 5.

Input:

```
0
```

Output:

```
CORRECT
```

Explanation:
Empty tree is considered correct.

Sample 6.

Input:

```
1
2147483647 -1 -1
```

Output:

```
CORRECT
```

Explanation:

The maximum possible value of the 32-bit integer type is allowed as key in the tree.

Sample 7.

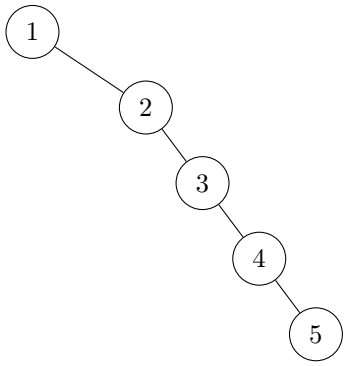
Input:

```
5
1 -1 1
2 -1 2
3 -1 3
4 -1 4
5 -1 -1
```

Output:

```
CORRECT
```

Explanation:



The tree doesn't have to be balanced. We only need to test whether it is a correct binary search tree, which the tree in this example is.

Sample 8.

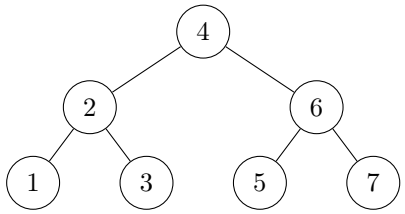
Input:

```
7
4 1 2
2 3 4
6 5 6
1 -1 -1
3 -1 -1
5 -1 -1
7 -1 -1
```

Output:

```
CORRECT
```

Explanation:



This is a full binary tree, and the property of the binary search tree is satisfied in every node.

Starter Files

The starter solutions for this problem read the input data from the standard input, pass it to a blank procedure, and then write the result to the standard output. You are supposed to implement your algorithm in this blank procedure if you are using `C++`, `Java`, or `Python3`. For other programming languages, you need to implement a solution from scratch. Filename: `is_bst_hard`

What to Do

Try to adapt the algorithm from the previous problem to the case when duplicate keys are allowed, and beware of integer overflow!

Need Help?

Ask a question or see the questions asked by other learners at this forum thread.

4 Advanced Problem: Set with range sums

We strongly recommend you start solving advanced problems only when you are done with the basic problems (for some advanced problems, algorithms are not covered in the video lectures and require additional ideas to be solved; for some other advanced problems, algorithms are covered in the lectures, but implementing them is a more challenging task than for other problems).

Problem Introduction

In this problem, your goal is to implement a data structure to store a set of integers and quickly compute range sums.

Problem Description

Task. Implement a data structure that stores a set S of integers with the following allowed operations:

- **add**(i) — add integer i into the set S (if it was there already, the set doesn't change).
- **del**(i) — remove integer i from the set S (if there was no such element, nothing happens).
- **find**(i) — check whether i is in the set S or not.
- **sum**(l, r) — output the sum of all elements v in S such that $l \le v \le r$.

Input Format. Initially the set S is empty. The first line contains n — the number of operations. The next n lines contain operations. Each operation is one of the following:

- "+ i" — which means **add** some integer (not i, see below) to S,
- "- i" — which means **del** some integer (not i, see below)from S,
- "? i" — which means **find** some integer (not i, see below)in S,
- "s l r" — which means compute the **sum** of all elements of S within some range of values (not from l to r, see below).

However, to make sure that your solution can work in an online fashion, each request will actually depend on the result of the last **sum** request. Denote $M = 1\ 000\ 000\ 001$. At any moment, let x be the result of the last **sum** operation, or just 0 if there were no **sum** operations before. Then

- "+ i" means **add**$((i + x) \bmod M)$,
- "- i" means **del**$((i + x) \bmod M)$,
- "? i" means **find**$((i + x) \bmod M)$,
- "s l r" means **sum**$((l + x) \bmod M, (r + x) \bmod M)$.

Constraints. $1 \le n \le 100\ 000$; $0 \le i \le 10^9$.

Output Format. For each find request, just output "Found" or "Not found" (without quotes; note that the first letter is capital) depending on whether $(i+x) \bmod M$ is in S or not. For each **sum** query, output the sum of all the values v in S such that $((l+x) \bmod M) \le v \le ((r+x) \bmod M)$ (it is guaranteed that in all the tests $((l + x) \bmod M) \le ((r + x) \bmod M)$), where x is the result of the last **sum** operation or 0 if there was no previous **sum** operation.

Time Limits.

language	C	C++	Java	Python	C#	Haskell	JavaScript	Ruby	Scala
time (sec)	1	1	4	120	1.5	2	120	120	4

Memory Limit. 512MB.

Sample 1.

Input:

```
15
? 1
+ 1
? 1
+ 2
s 1 2
+ 1000000000
? 1000000000
- 1000000000
? 1000000000
s 999999999 1000000000
- 2
? 2
- 0
+ 9
s 0 9
```

Output:

```
Not found
Found
3
Found
Not found
1
Not found
10
```

Explanation:
For the first 5 queries, $x = 0$. For the next 5 queries, $x = 3$. For the next 5 queries, $x = 1$. The actual list of operations is:
find(1)
add(1)
find(1)
add(2)
sum$(1, 2) \to 3$
add(2)
find$(2) \to$ Found
del(2)
find$(2) \to$ Not found
sum$(1, 2) \to 1$
del(3)
find$(3) \to$ Not found
del(1)
add(10)
sum$(1, 10) \to 10$

Adding the same element twice doesn't change the set. Attempts to remove an element which is not in the set are ignored.

Sample 2.

Input:

```
5
? 0
+ 0
? 0
- 0
? 0
```

Output:

```
Not found
Found
Not found
```

Explanation:
First, 0 is not in the set. Then it is added to the set. Then it is removed from the set.

Sample 3.

Input:

```
5
+ 491572259
? 491572259
? 899375874
s 310971296 877523306
+ 352411209
```

Output:

```
Found
Not found
491572259
```

Explanation:
First, 491572259 is added to the set, then it is found there. Number 899375874 is not in the set. The only number in the set is now 491572259, and it is in the range between 310971296 and 877523306, so the sum of all numbers in this range is equal to 491572259.

Starter Files

The starter solutions in C++, Java and Python3 read the input, write the output, fully implement splay tree and show how to use its methods to solve this problem, but don't solve the whole problem. You need to finish the implementation. If you use other languages, you need to implement a solution from scratch.

Note that we strongly encourage you to use stress testing, max tests, testing for min and max values of each parameter according to the restrictions section and other testing techniques and advanced advice from this reading. If you're stuck for a long time, you can read ths forum thread to find out what other learners struggled with, how did they overcome their troubles and what tests did they come up with. If you're still stuck, you can read the hints in the next What to Do section mentioning some of the common problems and how to test for them, resolve some of them. Finally, if none of this worked, we included some of the trickier test cases in the starter files for this problem, so you can use them to debug your program if it fails on one of those tests in the grader. However, you will learn more if you pass this problem without looking at those test cases in the starter files.

What to Do

Use splay tree to efficiently store the set, add, delete and find elements. For each node in the tree, store additionally the sum of all the elements in the subtree of this node. Don't forget to update this sum each

time the tree changes. Use split operation to cut ranges from the tree. To get the sum of all the needed elements after split, just look at the sum stored in the root of the splitted tree. Don't forget to merge the trees back after the **sum** operation.

Some hints based on the problems some learners encountered with their solutions:

- Use the sum attribute to keep updated the sum in the subtree, don't compute the sum from scratch each time, otherwise it will work too slow.
- Don't forget to do splay after each operation with a splay tree.
- Don't forget to splay the node which was accessed last during the find operation.
- Don't forget to update the root variable after each operation with the tree, because splay operation changes root, but it doesn't change where your root variable is pointing in some of the starters.
- Don't forget to merge back after splitting the tree.
- When you detach a node from its parent, don't forget to detach pointers from both ends.
- Don't forget to update all the pointers correctly when merging the trees together.
- Test sum operation when there are no elements within the range.
- Test sum operation when all the elements are within the range.
- Beware of integer overflow.
- Don't forget to check for null when erasing.
- Test: Try adding nodes in the tree in such an order that the tree becomes very unbalanced. Play with this visualization to find out how to do it. Create a very big unbalanced tree. Then try searching for an element that is not in the tree many times.
- Test: add some elements and then remove all the elements from the tree.

Need Help?

Ask a question or see the questions asked by other learners at this forum thread.

5 Advanced Problem: Rope

We strongly recommend you start solving advanced problems only when you are done with the basic problems (for some advanced problems, algorithms are not covered in the video lectures and require additional ideas to be solved; for some other advanced problems, algorithms are covered in the lectures, but implementing them is a more challenging task than for other problems).

Problem Introduction

In this problem you will implement Rope — data structure that can store a string and efficiently cut a part (a substring) of this string and insert it in a different position. This data structure can be enhanced to become persistent — that is, to allow access to the previous versions of the string. These properties make it a suitable choice for storing the text in text editors.

This is a very advanced problem, harder than all the previous advanced problems in this course. Don't be upset if it doesn't crack. Congratulations to all the learners who are able to successfully pass this problem!

Problem Description

Task. You are given a string S and you have to process n queries. Each query is described by three integers i, j, k and means to cut substring $S[i..j]$ (i and j are 0-based) from the string and then insert it after the k-th symbol of the remaining string (if the symbols are numbered from 1). If $k = 0$, $S[i..j]$ is inserted in the beginning. See the examples for further clarification.

Input Format. The first line contains the initial string S.
The second line contains the number of queries q.
Next q lines contain triples of integers i, j, k.

Constraints. S contains only lowercase english letters. $1 \le |S| \le 300\,000$; $1 \le q \le 100\,000$; $0 \le i \le j \le n - 1$; $0 \le k \le n - (j - i + 1)$.

Output Format. Output the string after all q queries.

Time Limits.

language	C	C++	Java	Python	C#	Haskell	JavaScript	Ruby	Scala
time (sec)	3	3	6	120	4.5	6	120	120	12

Memory Limit. 512MB.

Sample 1.

Input:

```
hlelowrold
2
1 1 2
6 6 7
```

Output:

```
helloworld
```

Explanation:
hlelowrold → *hellowrold* → *helloworld*
When $i = j = 1$, $S[i..j] = l$, and it is inserted after the 2-nd symbol of the remaining string *helowrold*, which gives *hellowrold*. Then $i = j = 6$, so $S[i..j] = r$, and it is inserted after the 7-th symbol of the remaining string *hellowold*, which gives *helloworld*.

Sample 2.

Input:

```
abcdef
2
0 1 1
4 5 0
```

Output:

```
efcabd
```

Explanation:
$abcdef \to cabdef \to efcabd$

Starter Files

The starter solutions for C++ and Java in this problem read the input, implement a naive algorithm to cut and paste substrings and write the output. The starter solution for Python3 just reads the input and writes the output. You need to implement a data structure to make the operations with string very fast. If you use other languages, you need to implement the solution from scratch.

What to Do

Use splay tree to store the string. Use the split and merge methods of the splay tree to cut and paste substrings. Think what should be stored as the key in the splay tree. Try to find analogies with the ideas from this lecture.

Need Help?

Ask a question or see the questions asked by other learners at this forum thread.