# Programming Assignment 1: Basic Data Structures

Revision: May 4, 2018

## Introduction

Welcome to your first Programming Assignment in the Data Structures Fundamentals course of the Algorithms and Data Structures MicroMasters program!

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 **??** and **??** 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.
5. Extend the standard stack interface with a new method.

## Passing Criteria: 2 out of 4

Passing this programming assignment requires passing at least 2 out of 4 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.**

| language | C | C++ | Java | Python | JavaScript | Scala |
|---|---|---|---|---|---|---|
| time (sec) | 1 | 1 | 1 | 1 | 3 | 3 |

**Memory Limit.** 512MB.

**Sample 1.**

Input:

```
[]
```

Output:

```
Success
```

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
```

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
```

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
```

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

**Sample 5.**

Input:

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

Output:

```
Success
```

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
```

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
```

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
```

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

**Sample 9.**

Input:

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

Output:

```
10
```

`)` 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.**

| language | C | C++ | Java | Python | JavaScript | Scala |
|---|---|---|---|---|---|---|
| time (sec) | 1 | 1 | 6 | 3 | 3 | 3 |

**Memory Limit.** 512MB.

**Sample 1.**

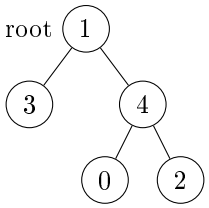
Input:

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

Output:

```
3
```

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:

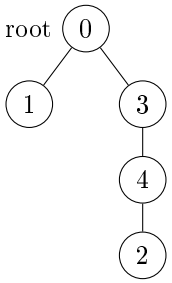
```
5
-1 0 4 0 3
```

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. $1 \le S \le 10^5$; $0 \le n \le 10^5$; $0 \le A_i \le 10^6$; $0 \le P_i \le 10^3$; $A_i \le A_{i+1}$ 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.**

| language | `C` | `C++` | `Java` | `Python` | `JavaScript` | `Scala` |
|---|---|---|---|---|---|---|
| time (sec) | 2 | 2 | 6 | 8 | 6 | 6 |

**Memory Limit.** 512MB.

**Sample 1.**

Input:

```
1 0
```

Output:

```

```

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

**Sample 2.**

Input:

```
1 1
0 0
```

Output:

```
0
```

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
```

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
```

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.

# 4 Solving a Programming Challenge in Five Easy Steps

## 4.1 Reading Problem Statement

Start by reading the problem statement that contains the description of a computational task, time and memory limits, and a few sample tests. Make sure you understand how an output matches an input in each sample case.

If time and memory limits are not specified explicitly in the problem statement, the following default values are used.

**Time Limits.**

| language | C | C++ | Java | Python | JavaScript | Scala |
|---|---|---|---|---|---|---|
| time (sec) | 1 | 1 | 1.5 | 5 | 5 | 3 |

**Memory limit:** 512 Mb.

## 4.2 Designing an Algorithm

After designing an algorithm, prove that it is correct and try to estimate its expected running time on the most complex inputs specified in the constraints section. If you laptop performs roughly $10^8$–$10^9$ operations per second, and the maximum size of a dataset in the problem description is $n = 10^5$, then an algorithm with quadratic running time is unlikely to fit into the time limit (since $n^2 = 10^{10}$), while a solution with running time $O(n \log n)$ will. However, an $O(n^2)$ solution will fit if $n = 1\,000$, and if $n = 100$, even an $O(n^3)$ solutions will fit. Although polynomial algorithms remain unknown for some hard problems in this book, a solution with $O(2^n n^2)$ running time will probably fit into the time limit as long as $n$ is smaller than 20.

## 4.3 Implementing an Algorithm

Start implementing your algorithm in one of the following programming languages supported by our automated grading system: `C`, `C++`, `Haskell`, `Java`, `JavaScript`, `Python2`, `Python3`, or `Scala`. For all problems, we provide starter solutions for `C++`, `Java`, and `Python3`. For other programming languages, you need to implement a solution from scratch. The grading system detects the programming language of your submission automatically, based on the extension of the submission file.

We have reference solutions in `C++`, `Java`, and `Python3` (that we don't share with you) which solve the problem correctly under the given constraints, and spend at most $1/3$ of the time limit and at most $1/2$ of the memory limit. You can also use other languages, and we've estimated the time limit multipliers for them. However, we have no guarantee that a correct solution for a particular problem running under the given time and memory constraints exists in any of those other languages.

In the Appendix, we list compiler versions and flags used by the grading system. We recommend using the same compiler flags when you test your solution locally. This will increase the chances that your program behaves in the same way on your machine and on the testing machine (note that a buggy program may behave differently when compiled by different compilers, or even by the same compiler with different flags).

## 4.4 Testing and Debugging

Submitting your implementation to the grading system without testing it first is a bad idea! Start with small datasets and make sure that your program produces correct results on all sample datasets. Then proceed to checking how long it takes to process a large dataset. To estimate the running time, it makes sense to implement your algorithm as a function like `solve(dataset)` and then implement an additional procedure `generate()` that produces a large dataset. For example, if an input to a problem is a sequence of integers of length $1 \le n \le 10^5$, then generate a sequence of length $10^5$, pass it to your `solve()` function, and ensure that the program outputs the result quickly.

Check the boundary values to ensure that your program processes correctly both short sequences (e.g., with 2 elements) and long sequences (e.g., with $10^5$ elements). If a sequence of integers from 0 to, let's say, $10^6$ is given as an input, check how your program behaves when it is given a sequence $0, 0, \ldots, 0$ or a sequence $10^6, 10^6, \ldots, 10^6$. Afterwards, check it also on randomly generated data. Check degenerate cases like an empty set, three points on a single line, a tree which consists of a single path of nodes, etc.

After it appears that your program works on all these tests, proceed to stress testing. Implement a slow, but simple and correct algorithm and check that two programs produce the same result (note however that this is not applicable to problems where the output is not unique). Generate random test cases as well as biased tests cases such as those with only small numbers or a small range of large numbers, strings containing a single letter "a" or only two different letters (as opposed to strings composed of all possible Latin letters), and so on. Think about other possible tests which could be peculiar in some sense. For example, if you are generating graphs, try generating trees, disconnected graphs, complete graphs, bipartite graphs, etc. If you generate trees, try generating paths, binary trees, stars, etc. If you are generating integers, try generating both prime and composite numbers.

## 4.5 Submitting to the Grading System

When you are done with testing, submit your program to the grading system! Go to the submission page, create a new submission, and upload a file with your program (make sure to upload a source file rather than an executable). The grading system then compiles your program and runs it on a set of carefully constructed tests to check that it outputs a correct result for all tests and that it fits into the time and memory limits. The grading usually takes less than a minute, but in rare cases, when the servers are overloaded, it might take longer. Please be patient. You can safely leave the page when your solution is uploaded.

As a result, you get a feedback message from the grading system. You want to see the "`Good job!`" message indicating that your program passed all the tests. The messages "`Wrong answer`", "`Time limit exceeded`", "`Memory limit exceeded`" notify you that your program failed due to one of these reasons. If

you program fails on one of the first two test cases, the grader will report this to you and will show you the test case and the output of your program. This is done to help you to get the input/output format right. In all other cases, the grader will *not* show you the test case where your program fails.

# 5 Appendix: Compiler Flags

**C** (`gcc 5.2.1`). File extensions: `.c`. Flags:

```
gcc -pipe -O2 -std=c11 <filename> -lm
```

**C++** (`g++ 5.2.1`). File extensions: `.cc`, `.cpp`. Flags:

```
g++ -pipe -O2 -std=c++14 <filename> -lm
```

If your `C/C++` compiler does not recognize `-std=c++14` flag, try replacing it with `-std=c++0x` flag or compiling without this flag at all (all starter solutions can be compiled without it). On Linux and MacOS, you most probably have the required compiler. On Windows, you may use your favorite compiler or install, e.g., `cygwin`.

**Java** (`Open JDK 8`). File extensions: `.java`. Flags:

```
javac -encoding UTF-8
java -Xmx1024m
```

**JavaScript** (`Node v6.3.0`). File extensions: `.js`. Flags:

```
nodejs
```

**Python 2** (`CPython 2.7`). File extensions: `.py2` or `.py` (a file ending in `.py` needs to have a first line which is a comment containing "python2"). No flags:

```
python2
```

**Python 3** (`CPython 3.4`). File extensions: `.py3` or `.py` (a file ending in `.py` needs to have a first line which is a comment containing "python3"). No flags:

```
python3
```

**Scala** (`Scala 2.11.6`). File extensions: `.scala`.

```
scalac
```