# Systems Programming – Assignment #1
Streaming Service System
2021/Fall

## 1 Before You Start

- It is mandatory to submit all the assignments in pairs. It is recommended to find a partner as soon as possible and create a submission group in the submission system. Once the submission deadline has passed, it will not be possible to create submission groups even if you have an approved extension.
- Read the assignment together with your partner and make sure you understand the tasks. Please do not ask questions before you have read the whole assignment.
- Skeleton classes will be provided on the assignment page, you must use these classes as a basis for your project and implement (at least) all the functions that are declared in them.

KEEP IN MIND

While you are free to develop your project on whatever environment you want, your project will be tested and graded ONLY on a CS LAB UNIX machine or on the VM image we supplied. It is your own responsibility to deliver a code that compiles, links and runs on it. Failure to do so will result in a grade 0 to your assignment.

Therefore, it is mandatory that you compile, link and run your assignment on a lab unix machine/VM image before submitting it.

We will reject, upfront, any appeal regarding this matter!!

We do not care if it runs on any other Unix/Windows machine.

Please remember, it is unpleasant for us, at least as it is for you, to fail your assignments, just do it the right way.

## 2 Assignment Goals

The objective of this assignment is to design an object-oriented system and gain implementation experience in C++ while using classes, standard data structures and unique C++ properties such as the “Rule of 5”. You will learn how to handle memory in C++ and avoid memory leaks. The resulting program must be as efficient as possible.

## 3 Assignment Definition

In this assignment you will help flattening the curve, by writing a C++ program that simulates a contact tracing system.  The program simulates a social network (represented as a graph), in which an epidemic is spreading, and contact tracers are searching for infected people, trace their contacts, and attempt to break the chain of infection.

The program will receive a config file (json) as an input, which includes the description of the social network graph, a list of agents (viruses and contact tracers), and additional settings (Fully described in section 3.5).

### 3.1 The Program flow

The program receives the path of the config file as the first command line argument. After the program is initialized, the function *simulate()* would trigger. The *simulate()* function runs in a loop, until the termination conditions (Which are detailed in section 3.6) are satisfied.

In each iteration of the loop (Called a *cycle*), each agent in the session acts on the graph.  The two types of agents which operate on the graph are contact tracers and viruses.  Viruses infect the node they occupy, and spread themselves (By creating new viruses) into adjacent nodes. Contact tracers attempt to break the chain of infection by disconnecting a node from the graph. The choice of the node to disconnect, is according to some pre-defined logic (Described in section 3.4).

For a detailed running example, see section 3.8.

### 3.2 Classes

**Session –** This is the main class, which holds the graph, the list of agents, and the algorithm the contact tracers use to decide which node to disconnect from the graph.

**Graph** – This class represents a graph, represented using an adjacency matrix. Each node in the graph has an index. You may assume that the graph is always undirected.

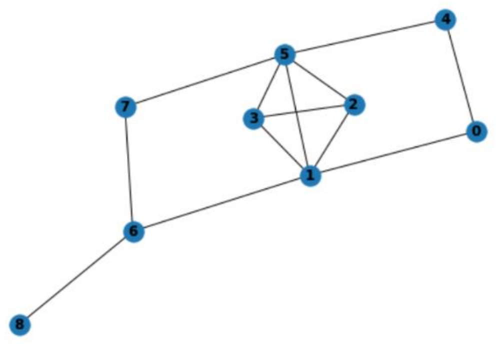


*Figure 1: An example graph*

**Agent** – An agent can be either a virus, or a contact tracer. It holds a reference to the main session. In each cycle, the agent affects the graph in a specific way  (described in the next section), using the pure virtual function act().

**Tree** – This class used to represent a shortest path tree in the graph, from some starting node. It is used by the contact tracers to decide which node to disconnect from the graph. It has a pure virtual method – traceContact(), which returns the index of the node in the graph, which the contact tracer should disconnect.

## 3.3 Agents

There are two types of agents active in the graph, Viruses and Contact Tracers. In each cycle the agents affect the graph in the following way:

**Virus –** Infects the node it occupies, if it is not already infected. The virus spreads itself in each iteration to one of the neighbors of the host node, in ascending order (By node indices).  For example, in the graph in Figure 1, consider a virus whose host is the node 0. In the first iteration, it will infect the host, and create a new virus in the node 1. Then in the second iteration, if the edge (0,4) still exists, it would create a copy of itself in the node 4.

**Contact Tracer –** Deques an infected node from infection queue, and created the shortest path tree from that node using BFS algorithm from the infected node. Then uses the method traceContact() to obtain an index of a node in the graph, and removes all the edges from the graph which are incident with this node.

**Important Notes:**

- The order with which the agents affect the graph is according to their time of addition to the session, where agents that were added first would act before agents that were added later. The agents that appear in the config file would be added according to the order with which they appear in the config file.
- Agents that were added in the current cycle won't act until the next cycle. For example – If in a cycle i, a virus v0 created a copy of itself v1, v1 would only act in the next cycle (that is – i+1).

## 3.4 Trees

Trees are created when running the BFS algorithm from one source node. Each tree contains the label of the root of the tree (the node in the graph it represents), and a vector of trees, which are his children. The children are ordered in the list from the one with the smallest label, to the one with the highest. The child with the smallest label is also termed *left-most*, while the child with the highest label is termed *right-most*.

The class tree also has a pure virtual method *traceContact()* which is used by the contact tracer to decide which node to disconnect from the graph.

There are three types of trees to implement in this assignment, these differ by their traceContact() method.

**RootTree** – Simply returns the index of the root of the tree.

**MaxRankTree** – Returns the index of the node in the tree with the highest number of children. In case of a tie, the node with the smallest depth in the tree would be picked. If there is still a tie, then the left-most node in that tree would be picked.

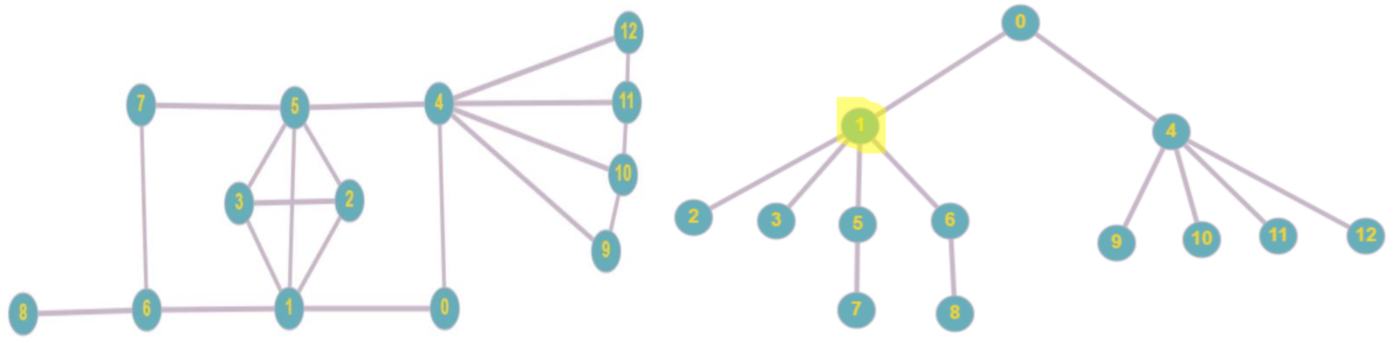


Figure 2: A graph, and the BFS tree created from the graph, with source node 0.

For example, consider the graph in figure 2, and a BFS tree created from the source node 0. If the tree is a maxRank Tree, the node which will be returned by the method *traceContact()* would be 1 (marked in yellow), as it has the maximal rank, together with 4, and the same depth as 4, but is further left in the tree then 4.

**CycleTree** – Starts traversing the tree from the root, picking always the left-most child. Returns the c'th node in this trip, where c is the cycle in which the tree was created. If the trip is less than c nodes long, returns the last node in it.

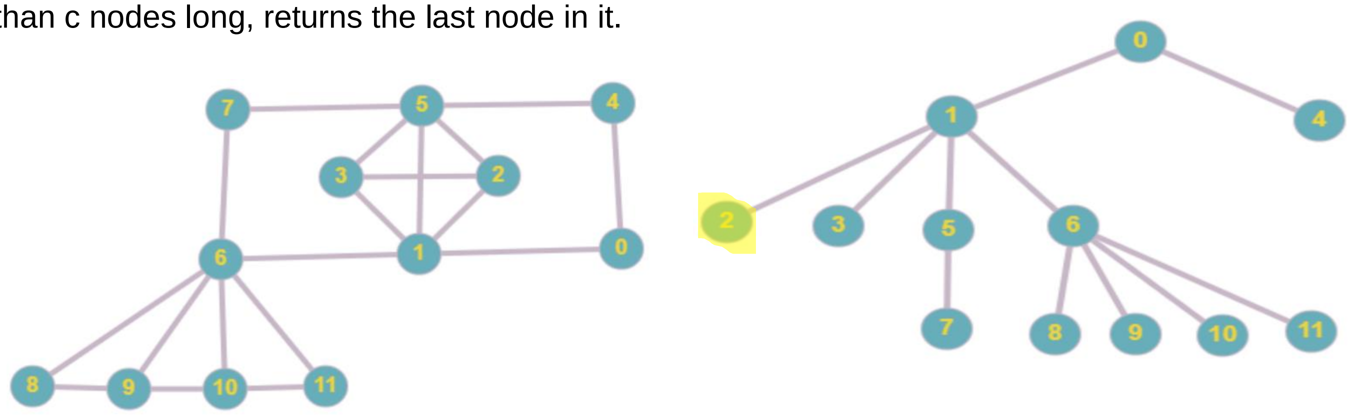


Figure 3: A graph and the BFS tree created from the source node 0.

For example, consider the graph and the tree in Figure 3.  Suppose that the tree was created in the third cycle (Cycle 2), then the node which will be returned by the method *traceContact()* is 2 as it is the third node in the mentioned trip.

**Important Note:**

When you implement the BFS algorithm the order with which you visit the neighbors is according to the indices, in ascending order. For example – if you visit a node with two neighbors, one indexed 1 and the other indexed 4, then the node indexed 1 would be visited before the node indexed 4.

## 3.5 Configuration file format

The configuration file is given in a JSON format, and it contains a dictionary (hash map) with the following entries:

**Graph** – The graph is given as an adjacency matrix.

**Agents** – A list of agents, each given as a pair of a string ("V" for virus, "C" for contact tracer) and the index of node (Which would be -1 for contact tracers, since they don't occupy a node in the graph).

**Tree** – The type of the trees to be constructed during the session, given as a string, "C" for cycle, "M" for max-rank, and "R" for root.

You may assume that the config file is valid, that is – it is a json file which contains the mentioned dictionary, each agent is described by a valid index and a valid string, etc.

In order to read JSON format with C++, we supplied you with Niels Lohman's JSON for Modern C++.  You can learn how to use this package, and see examples here:

https://github.com/nlohmann/json

## 3.6 Termination Conditions

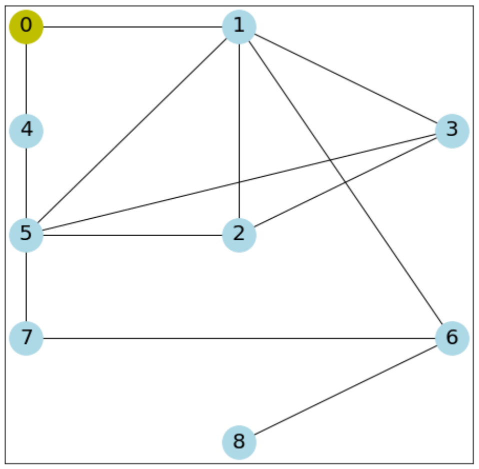
The program terminates when each connected component of the graph is either fully infected, or doesn't contain a virus in it.

## 3.7 Output

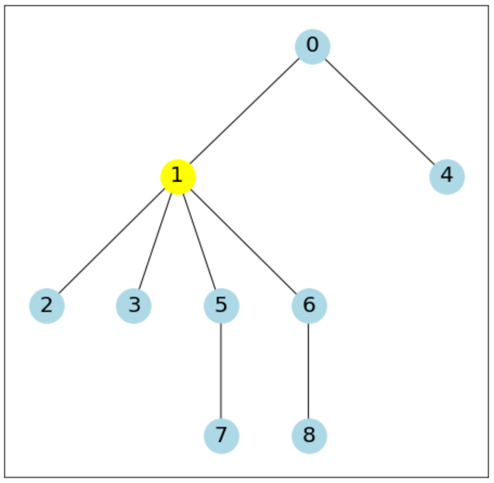
The program should create a file named output.json. The file should contain the graph in the last iteration, and a list of infected nodes.

## 3.8 Running Example

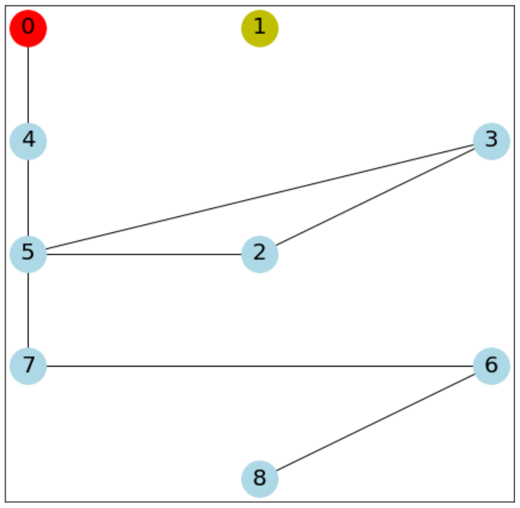
Consider the following graph, with a single virus in node 0, and a single contact tracer, using MaxRank trees. Blue nodes are healthy, yellow nodes carry a virus, and red nodes are sick.



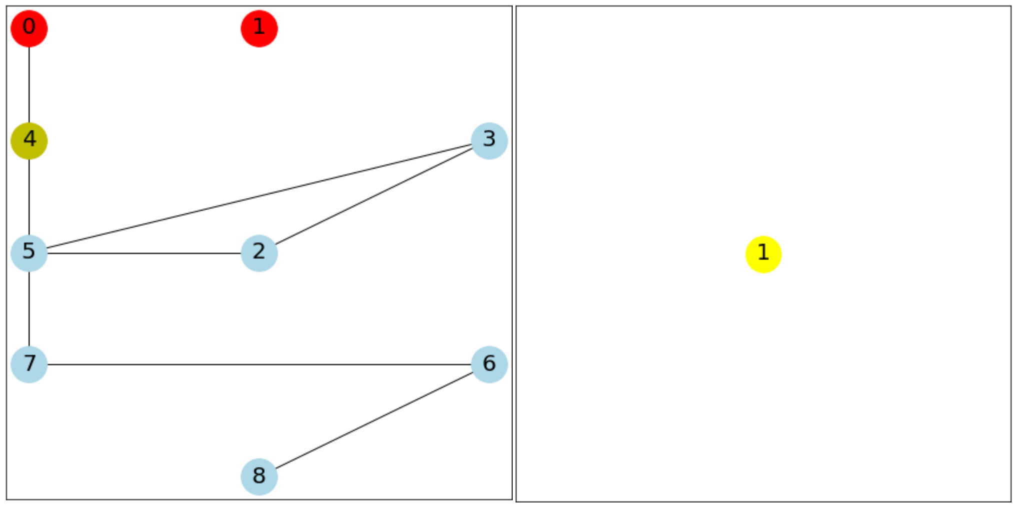
In the first iteration, the virus infects node 0, and spreads itself to node 1. The contact tracer, polls the node 0 from the infected queue, and creates a MaxRankTree whose root is the node 0.



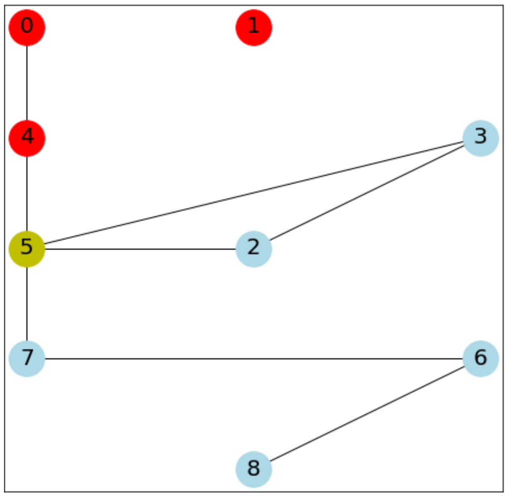
The node 1 is chosen, since it has the maximal rank, and hence all the edges incident to it are removed from the graph. After the first iteration, we obtain the following graph:



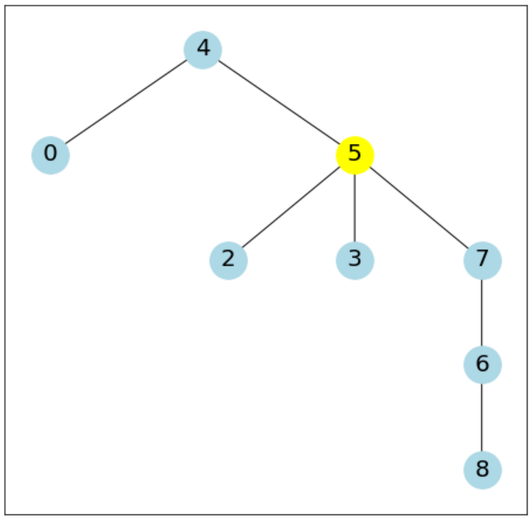
In the second iteration, the first virus spreads itself to node 4. The virus in node 1 infects node 1, but cannot spread itself further, since 1 is disconnected from the graph. The contact tracer then polls 1 from the infection queue, creates a BFS tree from 1 (Which contains only 1), and thus disconnects 1 from the graph (Which does nothing at that point).

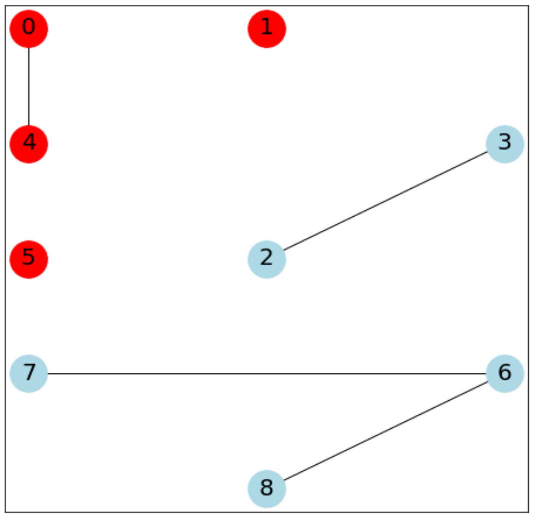


In the third iteration, 4 is infected, and the virus also spreads itself to node 5. Note that in this iteration the contact tracer does nothing, as it appears before the virus in the agents list, and the infection queue is still empty when it's his turn to act.



In the fourth iteration, the contact tracer polls 4 from the infection queue, and creates a *MaxRankTree* from it. It decides to disconnect 5 from the graph, as it has the maximal rank. Then, the virus infects 5, but it cannot spread itself any further. The program now terminates, as the termination conditions are satisfied. Each connected component is either fully infected, or doesn't have a virus in it.





### 3.9 General instructions

- ❖ Class with resources must implement the rule of five. **We will test your rule of five implementations for classes with resources, including your implementation of the move constructor, and the move assignment operator.**
- ❖ You may not change the interface of the objects. i.e. you may not modify the function declaration nor add any public/protected methods/data members. See section 4 for more details.
- ❖ You may not add any global variables.

## 4 Provided files

The following files will be provided for you on the assignment homepage:

Session.h

Agent.h

Graph.h

Tree.h

Main.cpp

You are required to implement the supplied functions and to add the Rule-of-five functions as needed. You are **NOT ALLOWED** to modify the signature (the declaration) of any of the supplied functions. We will use these functions to test your code, therefore any attempt to change their declaration might result in a compilation error and a major deduction of your grade.

You also **must not** add any global variables to the program.

## 5 Submission

- Your submission should be in a single zip file called ”student1ID-student2ID.zip”. The files in the zip should be set in the following structure:
    - src/
    - include/
    - bin/
    - makefile

  **src/** directory includes all .cpp files that are used in the assignment.
  **Include/** directory includes the header (.h or *.hpp) files that are used in the assignment.
  **bin/** directory should be empty, no need to submit binary files. It will be used to place the compiled file when checking your work.

- The makefile should compile the cpp files into the bin/ folder and create an executable named ”cTrace” and place it also in the bin/ folder.
- Your submission will be build (compile+link) by running the following commands: "make".
- Your submission will be tested by running your program with different scenarios.
- Your submission must compile without warnings or errors on the department computers.
- We will test your program using VALGRIND in order to ensure no memory leaks have occurred. We will use the following valgrind command:

  ```
  valgrind --leak-check=full --show-reachable=yes [program-name] [program parameters].
  ```

  The expected valgrind output is:

  ```
  definitely lost: 0 bytes in 0 blocks
  indirectly lost: 0 bytes in 0 blocks
  possibly lost: 0 bytes in 0 blocks
  suppressed: 0 bytes in 0 blocks
  ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
  ```

  We will ignore the following error only:

  still reachable: 72,704 bytes in 1 blocks (known issue with std). **We will not ignore** "still reachable" with different values than **72,704** bytes in **1** blocks

- Compiler commands must include the following flags:
  `-g -Wall -Weffc++ -std=c++11`.

## 6 Recommendations

1. Be sure to implement the rule-of-five as needed. We will check your code for correctness and performance.
2. After you submit your file to the submission system, re-download the file that you have just submitted, extract the files and check that it compiles on the university labs or on the supplied VM image. Failure to properly compile or run on these systems will result in a zero grade for the assignment.

בהצלחה!