Final Project

> In this final project, you will tackle two application-oriented problems concerning graphs and computer networks. Your task is to provide C++ solution classes and test cases for each problem. While these problems share similarities, they have distinct constraints. Your final grade will be based on the performance of your solutions and the quality of the test cases you contribute.

> Please note that this project may pose a challenge to average students. Hence, we urge you to thoroughly review this document and begin your work early. We will rigorously adhere to the grading policy outlined later in this document.

Introduction to Multicast Networks

Before delving into the specifics of the problems in this project, let's provide a brief overview of multicast computer networks. Detailed problem descriptions are presented in the **Project Specification** section.

Computer Networks

A computer network is a system in which computers are interconnected through links. Each link establishes a connection between two computers. In this project, we assume every connection link is bi-directional. This means that if there is a connection link between computers **A** and **B**, then computer **A** can communicate with computer **B**, and vice versa.

Based on the description provided, we can represent a computer network as an undirected graph denoted as **G = (V, E)**, where **V** represents the set of computers, and **E** represents the set of connection links.

Network Packets and Bandwidth

In a computer network **G**, communication between computers is achieved through the transmission of data packets. As long as **G** remains a connected graph, any computer **A** in the set **V** can send packets to any other computer **B** in **V**. The transmission of these packets through the network results in the generation of network traffic. The volume of network traffic between two computers, **A** and **B**, is directly proportional to the number of packets exchanged between them.

In practical terms, each physical connection link has its own maximum capacity, representing the maximum amount of traffic it can accommodate simultaneously. For each connection link **e** in the set **E**, we define the *bandwidth limit* denoted as $\mathbf{b_e}$, which signifies the maximum network traffic that can

pass through **e** at any given moment. When the bandwidth limit $\mathbf{b_e}$ is reached, additional traffic should not be introduced to the connection link, as doing so may result in packet loss.

Bandwidth, Path, and Transmission Costs

In the real world, computer connection links are provided and maintained by various vendors, such as Chunghwa Telecom ADSL service and FarEasTone 5G network service. Consequently, transmitting data packets through these connection links is not cost-free.

To simplify our model, we introduce the concept of the *bandwidth cost* associated with transmitting packets over a connection link. We represent the bandwidth cost for transmitting a unit of traffic through a connection link **e** in **E** as $\mathbf{c_e}$. It's essential to note that the bandwidth cost can vary for each connection link but is always a non-negative value.

When sending packets from computer **A** to computer **B**, the path may involve more than one connection link. To evaluate the cost associated with this path, we define the *path cost* as $\mathbf{c_p}$. This path cost is the sum of the bandwidth costs along the entire path, which can be expressed as $\mathbf{c_p}$ = sum($\mathbf{c_{e1}}$, $\mathbf{c_{e2}}$, ...).

Lastly, the overall cost of transmitting packets along the path is referred to as the *transmission cost*. This cost represents the total expenditure required to transmit all the packets from the source(s) to the destination(s).

Network Multicasting

In the previous section, we discussed the costs associated with transmitting packets in a computer network. Now, let's consider a scenario where we want to efficiently live stream content, such as a YouTube Live Stream, over the network, with one source but multiple destinations. How can we minimize the transmission cost in such a situation?

The solution lies in multicast communication. In a multicast setup, there is a single source computer (represented as **s**) and a set of destination computers (denoted as **D**). The fundamental idea behind multicasting is to construct a *Multicast Tree*, denoted as **MT**, where the root is the source **s**, and **D** constitutes a subset of the tree nodes. Each non-leaf node $\mathbf{d_i}$ in the tree forwards the packets it receives to all of its child nodes. This approach helps eliminate the transmission of duplicate packets over the same connection links, ultimately reducing transmission costs.

We use a tuple (**id**, **s**, **D**, **p**, **t**) to define a multicast request, $\mathbf{r_{id}}$. Here, **id** serves as a unique identifier for the multicast request, **p** is a Boolean value indicating whether a partial multicast tree (explained in the next section) is acceptable, and **t** represents the traffic size of this multicast. Given that there may be multiple possible multicast trees for $\mathbf{r_{id}}$, we introduce $\mathbf{MT_{id}}$ as the minimum transmission cost multicast tree. It's important to note that $\mathbf{MT_{id}}$ might not be unique.

Additionally, we define $\mathbf{ct_{id}}$ as the transmission cost for $\mathbf{r_{id}}$. This cost is computed by summing the bandwidth cost $\mathbf{c_e}$ associated with each link **e** in $\mathbf{MT_{id}}$, and then multiplying the result by **t**.

Partial, Full, and Successful Multicast Trees

Note that it is possible that some destination nodes may not be reachable if there are no paths from the source node to these destination nodes with excess bandwidth to carry the required traffic.

Therefore, a multicast tree that connects only a portion of the destination nodes to the source node is referred to as a **Partial Multicast Tree**, while the multicast tree that connects the source node to all destination nodes is called a **Full Multicast Tree**.

A multicast tree is considered a **Successful Multicast Tree** if it allows the source node to connect to either all **reachable** destination nodes when **p** is set to **true**, or to all destination nodes when **p** is set to **false**.

SDN

Traditionally, each computer in the network **G** maintains its own routing table to determine how to handle incoming packets. As a result, coordinating the actions of hundreds of millions of computers in **G** to construct multicast trees and efficiently deliver packets is a formidable challenge.

However, with the advent of software-defined networking (SDN), modern large data centers now employ a centralized routing table managed by a single entity known as the SDN controller. The SDN controller can dictate the actions that each computer should take upon receiving a new packet. In this context, multicasting emerges as an efficient method for distributing identical packets throughout the network.

In this project, your task is to implement a system resembling an SDN controller within an SDN network and establish multicast trees for all incoming requests.

Project Specifications

Problem 1

For this problem, you will work with a connected undirected graph **G** = (**V**, **E**), which represents a computer network initially with zero traffic. Your task is to implement an SDN controller with three essential operations for managing the network.

In this graph **G**, each vertex **v** in **V** represents a computer, and each edge **e** in **E** represents a connection link. These links have associated attributes: the available bandwidth $\mathbf{b_e}$, and the bandwidth cost $\mathbf{c_e}$. For this problem, we have **D** = **V**, meaning that every node, including the source node itself, in **G** is a destination node for the multicast request. In other words, the stream is intended to be broadcast to every node. Additionally, **p** = **true**, indicating that you have the discretion to determine which nodes are **not reachable**, and partial multicast trees can be deemed successful in this case.

The three SDN operations are described below:

1. *insert(**id**, **s**, **D**, **true**, **t**)*: This operation involves finding the minimum cost **successful multicast tree** for this streaming request $\mathbf{r_{id}}$ without altering previous route assignments. After the completion of the operation, return the updated network **G** with the adjusted edge available bandwidth values and $\mathbf{MT_{id}}$.
    - Note that for this operation, the streaming data of each request should be forwarded without splitting.
2. *stop(**id**)*: This operation should perform the following three tasks sequentially to stop the streaming of a previous request that matches the provided id:
   (1). Remove the corresponding multicast assignment $\mathbf{r_{id}}$ from the network and release the bandwidth it consumed on each connection link.
   (2). After releasing the bandwidth, some connection links might become available for previously unreachable endpoints. Starting from the lowest request id, connect as many previously unreachable endpoints as possible to the partial multicast tree of each request with minimal additional transmission cost while keeping the previously assigned routes.
   (3). Return the updated network graph **G** with adjusted edge available bandwidth values and a list of all updated multicast trees sorted in ascending id order.
3. *rearrange()*: Reset the edge available bandwidth and re-insert all active multicast requests in ascending id order that have not been stopped. Return the updated network graph **G** with adjusted edge available bandwidth values and a list of all updated multicast trees for active requests sorted in ascending id order.

Each test case consists of the network graph **G** and subsequent operation calls. Your code should perform each operation call and return the correct answer. For detailed test case format and grading policy, please refer to the sections **Implementation & Test Cases** and **Score Calculation & Grading Policy**.

Problem 2

For this problem, your task is to design an effective heuristic algorithm that minimizes the total **penalty** associated with unsatisfied streaming traffic requests plus the transmission cost $\mathbf{ct_{id}}$ for each satisfied requests. The input is similar to Problem 1, with the exception that, for each request, **D** is a subset of **V**, meaning that the destination nodes might not include every node as in Problem 1. Additionally, **p** = **false**, indicating that all destination nodes must be connected, and only full multicast trees are considered successful. If unsuccessful or if not all destination nodes can be connected, no traffic assignments will be made, and a penalty value for this unconnected request will be collected at the end for the evaluation of your algorithm.

The three required operations for this problem are as follows:

1. *insert(**id**, **s**, **D**, **false**, **t**)*: Find a feasible, not necessarily minimum cost, **full multicast tree** for the request without altering previous route assignments. After the operation, return the updated network graph **G** with adjusted edge available bandwidth values and $\mathbf{MT_{id}}$.
    - Note that for this operation, the streaming data of each request should be forwarded without splitting.
2. *stop(**id**)*: This operation is essentially the same as the stop operation in Problem 1. Unsatisfied requests **may**, but is not obligated to, be satisfied once link bandwidths are released. You may build new multicast trees in any order. After the operation, return the updated network graph **G** with adjusted edge available bandwidth values and a list of all newly successful multicast trees sorted in ascending id order.
3. *rearrange()*: This operation **can** remove some successful multicast trees from the network and release the bandwidth. Then, rebuild the multicast trees for the removed active requests. Your algorithm can decide the order to rebuild the multicast trees, but you may not create **new** unsatisfied multicast requests during this operation. Your algorithm **can** also build new multicast trees for previously unsatisfied multicast requests. Return the updated network graph **G** with adjusted edge traffic values and a list of all updated multicast trees for active requests sorted in ascending id order.

Each test case for this problem consists of a netwok graph **G** and several operation calls. Your code should perform each operation call and return your answer. For a detailed test case format, penalty calculation, and grading policy, please refer to the sections **Implementation & Test Cases** and **Score Calculation & Grading Policy**.

Implementation & Test Cases

Overview

For this project, you will need to write two C++ classes. Each class should include specific public instance methods. The TA's program will call these public instance methods and check the return value. Each function call represents an operation defined in the problem definition section, and the returned value will be considered the answer you provide.

Each test case is a series of function calls. Therefore, your code will not know the whole test case in advance. Please keep this in mind when implementing your code.

Core Data Structures

Here we define the basic data structures for your program to use as function input and output parameters. These data structures are provided in basicDS.h for you. Please follow the format during

implementation strictly and **DO NOT** modify basicDS.h file. If you need additional data structures, you may write it in Problem1.h and / or Problem2.h header files.

```
// a mathematical set
struct Set {
    int size = 0;
    vector<int> destinationVertices;
};

// an edge in graph
struct graphEdge {
    int vertex[2];
    int b;          // remaining bandwidth
    int be;         // bandwidth limit
    int ce;         // bandwidth cost
};

// an edge in tree
struct treeEdge {
    int vertex[2];
};

// an undirected graph
struct Graph {
    vector<int> V;
    vector<graphEdge> E;
};

// a multicast tree
struct Tree {
    vector<int> V;
    vector<treeEdge> E;
    int s;          // source node
    int id;         // corresponding multicast request id
    int ct;         // transmission cost of multicast tree
};

// a forest
struct Forest {
    int size = 0;
    vector<Tree> trees;
};
```

Problem 1 Class & Public Instance Method Format

In your *Problem1* class, you should implement a constructor, a destructor and 3 additional public methods. The input parameters and return type should follow the definitions shown below. To output your answer of each operation, modify the call-by-reference parameters (*G*, *MTid* and *MTidForest*)

directly. The values stored in these parameters after each function returns will be viewed as your answer of each operation.

```
class Problem1 {
public:

    Problem1(Graph G);  //constructor
    ~Problem1();        //destructor
    void insert(int id, int s, Set D, int t, Graph &G, Tree &MTid);
    void stop(int id, Graph &G, Forest &MTidForest);
    void rearrange(Graph &G, Forest &MTidForest);
}
```

Problem 2 Class & Public Instance Method Format

Similar to Problem 1, implement a class named *Problem2* and total 5 public functions, including the constructor and destructor. The input and output mechanism is similar, with the only difference being the return type of *insert()* function. If the insert request is satisfied, return Boolean **true**, otherwise return Boolean **false**.

```
class Problem2 {
public:

    Problem2(Graph G); //constructor
    ~Problem2(); //destructor
    bool insert(int id, int s, Set D, int t, Graph &G, Tree &MTid);
    void stop(int id, Graph &G, Forest &MTidForest);
    void rearrange(Graph &G, Forest &MTidForest);
}
```

In addition to the 5 mandatory public functions of class *Problem1* and *Problem2*, you are free to include other variables or functions. You may also define other classes or structures when implementing, as long as they are properly declared and defined. For submission details, please refer to the **Submission Format** section.

Function and Library Limitations

Your program should not use the following libraries or their equivalent C counterparts:

1. Input/output library (including iostream, fstream...)
2. Filesystem library (including filesystem...)
3. Thread support library (including mutex, semaphore, thread...)

Your program should not use the following functions or their equivalent C counterparts:

1. *main()*: adding a *main()* will result in a compiler error
2. *system()*: no *system()* calls for security reason
3. *fork()*: no multiprocessing techniques

The TA's program will check your code automatically before compiling it and running it. Any rule violation or cyber attack attempt will be **considered cheating**.

Aside from the above restrictions, you are free to use any other standard libraries when implementing the final project, including the STL. The TAs' computers have installed all standard C++ libraries listed on the cppreference website (https://en.cppreference.com/w/cpp/header).

Compiler, C++ Version and Operating System

Your submitted code will be compiled following the C++20 standard using GNU G++ compiler version 9.4.0. The commands to compile your code along with TA's testing program (main1.cpp, main2.cpp) will be:

```
For Problem 1:
g++ -o main1 main1.cpp -std=c++2a

For Problem 2:
g++ -o main2 main2.cpp -std=c++2a
```

The TAs will finally run the executable file "main1" and "main2" to start testing and grading your code.

To assure fairness and to prevent operating system issues affecting your program performance or output, the TAs will use the open source Ubuntu 20.04.5 LTS operating system to compile and run your code on our Lab server. The Ubuntu operating system is free and open-sourced for everyone to download, you are encouraged to use a VMWare or Virtual Box to install it as a VM on your computer.

Test Case Parameter Size

The parameter size of each test cases, contributed by TAs or students, will follow these constraints:

```
1 <= |V| <= 100
1 <= |E| <= |V| * (|V| - 1)
1 <= t <= b <= 100
1 <= c <= 100
2 <= fc <= 100,000
```

- |V| and |E| represents the number of vertices (V) and edges (E) in the Graph G respectively.
- b and c represents $\mathbf{b_e}$ and $\mathbf{c_e}$ respectively.
- t represents the traffic size **t** of a multicast request.

- fc represents the number of function calls, including constructor and destructor

Score Calculation & Grading Policy

Your final project grade consists of four parts: (1) Problem 1 score, (2) Problem 2 score, (3) test case contribution, and (4) report. Each part has its weight, and the weighted average will be the final score for this project.

Problem 1 Score

weight = 0.35

Answer Correctness

Each test case consists of a sequence of function calls to your public functions in the *Problem1* class. The sequence will always start with the *Problem1()* constructor and end with the *~Problem1()* destructor. The TA's program will call your functions in the order of the sequence and immediately check the correctness of the returned values.

To determine the correctness of *insert()*, we will consider the following criteria:

1. The output $\mathbf{MT_{id}}$ is a tree containing **s** (not a forest) without cycles.
2. Each edge in $\mathbf{MT_{id}}$ has sufficient bandwidth before the multicast stream starts.
3. If $\mathbf{MT_{id}}$ is not a full multicast tree, it includes all reachable destinations.
4. $\mathbf{MT_{id}}$ has the minimum possible transmission cost.
5. The cost $\mathbf{ct_{id}}$ is calculated correctly.
6. The attributes of the network **G** after adding $\mathbf{MT_{id}}$ are correct.

To determine the correctness of *stop()*, we will consider the following criteria:

1. The output *MTidForest* is composed of several $\mathbf{MT_{id}}$, sorted in ascending id order.
2. Each edge in $\mathbf{MT_{id}}$ has sufficient bandwidth before the multicast stream starts.
3. If $\mathbf{MT_{id}}$ is not a full multicast tree, it includes all reachable destinations.
4. $\mathbf{MT_{id}}$ uses least additional transmission cost when connecting newly reachable destinations.
5. The cost $\mathbf{ct_{id}}$ is calculated correctly.
6. No existing routes are modified during *stop()* without permission.
7. The attributes of the network **G** after adding $\mathbf{MT_{id}}$ are correct.

To determine the correctness of *rearrange()*, we will consider the following criteria:

1. The output *MTidForest* is composed of all active multicast trees sorted in ascending id order.
2. Each edge in each $\mathbf{MT_{id}}$ has sufficient bandwidth before the multicast stream starts.

3. If $\mathbf{MT_{id}}$ is not a full multicast tree, it includes all reachable destinations.
4. $\mathbf{MT_{id}}$ has the minimum possible transmission cost.
5. The cost $\mathbf{ct_{id}}$ is calculated correctly.
6. The attributes of the network **G** after adding $\mathbf{MT_{id}}$ are correct.

If the return value of your function is incorrect at any point, the test case is considered **failed**, and the test case will stop immediately. The *~Problem1* destructor will be called.

Resource Limitation

For each test case, your program could use a maximum of 5 seconds of CPU and 500MB of memory on the TA's lab server. If your function consumes excessive CPU time or memory resources during execution, the test case will be considered to have exceeded its resource limits (RLE). In such cases, the test case will terminate immediately, and the *~Problem1* destructor will be called.

To help you estimate the CPU time of your program, we have provided a benchmark.cpp for you. When compiled with g++ without any optimization flags, it takes 32.38 seconds of user CPU time to finish execution on the TA's lab server.

Successful Test Case

If the test case is not **failed** or **RLE**, then it is considered **successful**, otherwise it is considered **unsuccessful**.

Problem 1 Point Calculation

Starting from 100 points, we will deduct 1 point for each **unsuccessful** test case. The remaining point is the score you get for this problem. The total test case will be less than 100, so you always get positive score for this problem.

Problem 2 Score

weight = 0.4

In each test case, you will have a sequence of function calls to your public functions in *Problem2*. This sequence begins with the *Problem2()* constructor and ends with the *~Problem2()* destructor. Unlike Problem 1, Problem 2 does not have a unique or correct answer. Instead, your program will be evaluated based on its performance relative to other students' programs.

Your program's performance will be measured using a raw score (**rs**), which is calculated as **rs = penalty + transmission cost**. The objective is to minimize **rs** for each test case, meaning you want to minimize both the penalty and transmission cost to achieve the highest possible score.

Penalty Calculation

Starting from 0, let's assume that 1 unit of virtual time passes between each function call in the test case sequence. For each unsatisfied multicast request $\mathbf{r_{id}}$ = (**id**, **s**, **D**, **false**, **t**), the penalty $\mathbf{p_{id}}$ is calculated as:

$\mathbf{p_{id}} = (\mathbf{start(i)} - \mathbf{end(i)})^2 * \mathbf{t}$

Here, **start(i)** represents the virtual time when $\mathbf{r_{id}}$ starts to be unsatisfied, and **end(i)** is the time when $\mathbf{r_{id}}$ is finally satisfied. If the test case sequence ends before $\mathbf{r_{id}}$ is satisfied, then **end(i)** will be the length of the test case sequence. Also, if a *stop(id)* is called before $\mathbf{r_{id}}$ is satisfied, then **end(i)** will be the time *stop(id)* is called.

For example, let's say we have an *insert()* call as the 6th function call to insert $\mathbf{r_5}$ = (5, 1, {3, 4}, **false**, 15). If $\mathbf{r_5}$ remains unsatisfied until a *stop()* call is made as the 31st function call, and finally $\mathbf{r_5}$ is satisfied, then the penalty will be calculated as $(\mathbf{30} - \mathbf{5})^2 * \mathbf{15}$ = 9375.

Transmission Cost

The transmission cost of each satisfied $\mathbf{r_{id}}$ = (**id**, **s**, **D**, **false**, **t**) is $\mathbf{ct_{id}}$. Transmission cost for each request is only counted once when the request is satisifed.

Following the example in previous section, assume $\mathbf{ct_{id}}$ is 300 when $\mathbf{r_5}$ is satisfied at the 31st function call. Then $\mathbf{r_5}$ contributed a total 9375 + 300 = 9675 to the **rs** of this test case.

Rules of the Operations

Your functions should not violate the rules of the operation. We will check for rule violations through the following criteria:

1. For each $\mathbf{MT_{id}}$ you returned, we will check:
   (1) $\mathbf{MT_{id}}$ is a full multicast tree.
   (2) The transmission cost is calculated correctly.
   (3) Each edge in $\mathbf{MT_{id}}$ has sufficient bandwidth.

2. For each **G** you returned, we will check:
   (1) The remaining bandwidth is correct.
   (2) The topology is correct.
   (3) Routes of previously routed multicasts are not changed unless permitted.

3. For each *MTidForest* you returned, we will check:
   (1) Each tree is a valid $\mathbf{MT_{id}}$
   (2) The trees are sorted in ascending id order

If any violation is found, the raw score **rs** for the test case will be set to INT_MAX, the *~Problem2()* destructor will be called and the test case will stop immediately.

Resource Limitation

For each test case, your program could use a maximum of 5 seconds of CPU and 500MB of memory on the TA's lab server. If your function consumes excessive CPU time or memory resources during execution, the test case will be considered to have exceeded its resource limits (RLE). In such cases, the raw score **rs** of the test case will be set to **INT64_MAX**, the test case will stop immediately, and the *~Problem2* destructor will be called.

To help you estimate the CPU time of your program, we have provided a benchmark.cpp for you. When compiled with g++ without any optimization flags, it takes 32.38 seconds of user CPU time to finish execution on the TA's lab server.

Raw Score and Normalized z-Score

For each test case, we calculate a raw score (**rs**) by summing up the penalty and transmission cost. To compare your performance with other students, we compute a normalized score (**z-score**) for each test case using the definition of the z-score in Statistics. We then average your **z-scores** over all the test cases to obtain a **z'-score**, which will be used to evaluate your algorithm for Problem 2.

Problem 2 Point Calculation

The score you receive for Problem 2 will be calculated as follows:

```
max(0, 88 - 9 * z')
```

In general, a smaller raw score implies a higher score for Problem 2. It's important to note that you will not receive a negative score for this problem, but there is a chance for you to earn more than 100 points here.

Test Case Contribution Score

weight = 0.15

Each student is required to provide two test cases for this project, one for each problem. These test cases should be saved in text files and must adhere to the format outlined in the following sections titled **Test Case Contribution Format** and **Implementation & Test Cases**.

Problem 1 Test Case Requirements

A valid Problem 1 test case should satisfy the following conditions:

1. The network graph **G** provided should be a connected undirected graph. Each vertex **v** in **V** should have a unique id, starting from 1 to **|V|**.
2. Each multicast request should have **D = V**. Furthermore, each request's **id** should be unique.
3. The parameters of the network graph **G** and each multicast request, including bandwidth limits, bandwidth costs, and traffic sizes, should adhere to the constraints outlined in the

Implementation & Test Cases section.

If your Problem 1 test case is valid, you earn 30 points, while you may earn up to 40 extra points based on the test case's level of difficulty. Test case difficulty is evaluated by comparing the average score of other students' programs. The most challenging test case will receive 30 points, while the least difficult one will get 0 points.

Problem 2 Test Case Requirements

A valid Problem 2 test case must meet the same requirements as Problem 1 test cases, with the following difference: **D** should be a subset of **V**.

The criteria for test case validity and difficulty scoring in Problem 2 are the same as those in Problem 1.

Test Case Contribution Format

Please name the files as **Problem1_test_case.txt** and **Problem2_test_case.txt**.

The content of each file should follow the below formats:

1. Begin with the graph definition. The graph definition should follow the format below:
    i. The first line contains two integers **|V|** and **|E|**, representing the number of vertices (V) and edges (E) in the Graph G. Add a whitespace between the two numbers.
    ii. The next **|E|** lines each represents the information of an edge between vertex **u** and **v** and has the format **u v** $\mathbf{b_e}$ $\mathbf{c_e}$. The vertex id starts with 1, ends with |E|, and should be a unique integer value.
2. Leave a blank line.
3. Then, according to the function call sequence you designed, specify the function names and paramters using the following format. Please separate function names and each parameter by a single white space.
    - insert **id s D t**
    - stop **id**
    - rearrange

Here's an example of Problem1_test_case.txt:

```
5 7
1 2 10 5
1 3 20 8
2 3 15 6
2 4 25 10
3 4 30 12
3 5 15 6
4 5 20 8
```

```
insert 1 2 { 1 2 3 4 5 } 10
insert 2 3 { 1 2 3 4 5 } 5
insert 3 4 { 1 2 3 4 5 } 15
stop 2
insert 4 1 { 1 2 3 4 5 } 5
rearrange
```

The above Graph G is depicted as follows:

(The bandwidth limit and cost for each edge is represented by $\mathbf{b_e}$ / $\mathbf{c_e}$)

Please ensure that your test cases adhere to the specified format and each parameter follows the constraints listed in section **Implementation & Test Cases**.

Report

weight = 0.10

Your report should include the following sections:

1. **Approach to Problem 1:** Describe how you approached solving Problem 1. Explain the algorithms and data structures you used. Provide the time and space complexities of each algorithm you implemented.
2. **Class(es) for Problem 1:** List the classes you used to solve Problem 1 and specify the functionality of each public and private method. Describe the input and output types of each method and explain the purpose of each instance method.
3. **Approach to Problem 2:** Explain your approach to solving Problem 2. Describe the algorithms and data structures you employed and provide the time and space complexities of these algorithms. Explain how your solution, especially how you select the connection links, optimizes the results.
4. **Class(es) for Problem 2:** List the classes you used for Problem 2 and specify the functionality of their public and private methods. Detail the input and output types of each method and explain the purpose of each instance method.
5. **Test Case Design:** Explain how you designed and created the test cases for both problems. Provide arguments supporting why you believe these test cases are of high quality.
6. **References:** List any papers, journals, websites, or materials you referenced to complete this final project. Include proper citations for any external sources you consulted.

Code of Conduct

The teacher and the TAs have established several crucial rules for this course. Please adhere to these rules while working on the final project:

1. **NO PLAGIARISM**: Honesty and academic integrity are highly valued. Any form of cheating, including plagiarism, will result in immediate expulsion from the course. Plagiarism detection tools will be employed to thoroughly examine your program and report. These tools are highly effective, and even renaming variable names or rephrasing sentences will still be detected.

2. **NO LATE SUBMISSION**: The final project deadline is non-negotiable. Late submissions will not be accepted under any circumstances. It is strongly advised to manage your time effectively and commence work on the project well in advance. Attempting to complete the project during the last week of the semester is considered impractical by the TAs.

3. **RESPECT COPYRIGHT**: Do not utilize pirated or unauthorized sources. If you include figures, algorithms, or charts created by others in your report, be sure to provide proper references to acknowledge the sources and respect copyright regulations.

Submission Format

Please submit the following files:

1. Two solution files for the problems named Problem1.cpp and Problem2.cpp.
2. Two test case files for the problems named Problem1_test_case.txt and Problem2_test_case.txt.
3. A report in PDF format named Report.pdf.
4. (Optional) Header files you created for the problems, named Problem1.h and Problem2.h.

Please compress all the above files into a single zip file named *{student id}_final_project.zip*, for example, 111061234_final_project.zip. Organize the files as follows:

```
{student id}_final_project.zip
    ├ Problem1.cpp
    ├ Problem2.cpp
    ├ Problem1.h (Optional)
    ├ Problem2.h (Optional)
    ├ Problem1_test_case.txt
    ├ Problem2_test_case.txt
    └ Report.pdf
```

Notes

Multicast in computer networks has been a subject of study for the past 30 years, resulting in numerous research papers and techniques available on the Internet. We encourage you to explore

existing methods and approaches from the Internet, as this can greatly contribute to enhancing your problem-solving skills.

For Problem 1 in our final project, each operation call is designed to have a correct answer. However, in Problem 2, there is no single unique and correct solution. Even today, some of the world's top network researchers are still striving to discover optimal solutions. Therefore, while the TAs can provide explanations and clarifications regarding the problem, it ultimately depends on your creativity and knowledge to devise the best solution.

Hints for Problem 1

1. Given that **D** is equal to **V**, this problem essentially involves finding the most cost-effective way to connect the entire graph **G**.

2. In contrast to the typical undirected graphs discussed in our lectures, each edge in **G** has a bandwidth limit. Consequently, during the execution of *insert()*, it might be necessary to initially create a temporary graph **G'** = (**V**, **E'**) for the request $\mathbf{r_{id}}$ = (**id**, **s**, **D**, **true**, **t**). This temporary graph **G'** only includes edges with remaining bandwidth no less than **t**. It's worth noting that this temporary graph **G'** may be disconnected, and the objective is to establish connections from as many nodes in **D** to **s** as possible at the lowest cost.

Hints for Problem 2

1. When **D** is a subset of **V**, the problem of finding the minimum cost tree connecting **s** to all vertices in **D** is commonly referred to as the **Steiner Tree** problem. It's worth noting that the Steiner Tree problem is a well-known NP-hard problem, implying that no efficient polynomial time and space algorithm exists to solve it. Furthermore, verifying whether a claimed solution is optimal or not is also an NP-hard task.

2. Over the past four decades, approximate and heuristic algorithms have been proposed to tackle the Steiner Tree problem. An influential approximate algorithm for the Steiner Tree problem was introduced by Kou et al. in 1981. This algorithm can provide good approximate solutions in polynomial time. We strongly recommend reading their concise 5-page paper as it offers valuable insights. Building on the work of Kou, many others have proposed their solutions to the problem, and you can reference their work when working on your final project.

3. Problem 2 isn't a straightforward Steiner Tree problem because it also takes into account the bandwidth limit of each edge. As in Problem 1, you may need to create a temporary graph **G'** = (**V**, **E'**) for the multicast request $\mathbf{r_{id}}$ = (**id**, **s**, **D**, **false**, **t**), containing only edges with remaining bandwidth not less than **t**.

4. It's crucial to note that, when performing the *insert()* operation, the penalty can accumulate rapidly and significantly impact the scores, particularly in cases where the total test case sequence is extensive. Therefore, you may want to avoid naively selecting solutions with the

lowest transmission cost during *insert()*. Instead, consider allocating bandwidth resources thoughtfully and assign a higher edge weight to an edge **e** in **E'** if the remaining bandwidth of **e** is limited.

Useful Web Sources

1. https://ieeexplore.ieee.org/Xplore/home.jsp: IEEE Xplore provides access to a vast collection of IEEE journals and papers. You can explore and discover research papers related to our final projects on this platform. Full papers can be accessed and downloaded using NTHU IP addresses.

2. https://dl.acm.org: The ACM Digital Library is home to ACM journals and papers. You can find valuable research papers related to our final projects here. Full papers are available for viewing and download using NTHU IP addresses.

3. https://ieeexplore.ieee.org/document/7980006: A research paper by Xu et al. The **Cost Model** section of this paper may provide valuable insights and inspiration when addressing Problem 2.

Reminders

1. This project is conceivably challenging, so start working on it as soon as possible!
2. The TAs also need to do lots of coding for their research works and find it very useful to look up the Internet for syntax questions and algorithm ideas. You may try to effectively utilize web resources to assist your project implementation. However, as we have stated clearly in the **Grading Policy** section, plagiarism is **STRICTLY PROHIBITED**. Here are some rules to follow:
    i. Searching for syntax and code online is encouraged, but don't do direct **copy and paste**. Understand the idea and **implement it in your own way**.
    ii. Feel free to discuss the problems, solutions, or ideas with your peers. However, **never give your code**, Google Drive, or GitHub repositories to them to avoid potential issues.
    iii. Study published research papers and learn from their GitHub repositories (if publicly accessible). However, don't download, clone, or fork their repositories; always implement on your own.
    iv. ChatGPT (or other generative AI tools) may help answering some syntax questions, but it is not always correct. You can reference it when writing your code, but don't use it to write your report. Please write the report by yourself.
3. The teacher and TAs have tried their best to design and review this project description, but they are also humans and might make mistakes. In case you find any unclear or obscure description, generally you should first read the definitions and descriptions twice. If, after your effort, it still confuses you, then do feel free to open a discussion thread **publicly on the eeClass system**. The TAs will respond within 24 hours. If a correction is necessary, the TAs will announce it to all students.
4. You are more than welcome to discuss your thoughts and solutions with the TAs during their office hours by making appointments via email in advance. However, since the TAs have limited

time for many duties, they cannot help with code debugging.

5. Please utilize the resourceful Internet for most questions and answers. Online tutorials are often better than the TAs' explanations.

Which TA Is Best to Answer Your Questions?

- TA Peter (陳唯中): The main designer of the final project and the head TA. You can ask/discuss with him about...
  - Algorithm and data structure suggestions
  - Solution ideas and possible approaches
  - Network and multicasting
- TA Kevin (彭冠文): The designer of grading and testing environments. You can ask/discuss with him about...
  - Problem 1 and Problem 2 public instance method parameters and return types
  - Time and memory limitations
  - Test case generation
- TA Alice (簡映榕): The designer of grading and testing environments. You can ask/discuss with her about...
  - Problem 1 and Problem 2 public instance method parameters and return types
  - Z-score and project score calculations
  - Plagiarism checks