EE 205 Data Structures and Algorithms **Fall 2018**

Project 2: Receiving Messages from a Packet Network

Lecturer: Yung Yi *TA: Kyunghwan Son*

2.1 Introduction

Modern computer networks take messages and break them into smaller units called packets. Because of redundancy, dynamic routing techniques, and other networking mechanisms, packets often arrive at their destination completely out of order. The end node computer must put these packets in the correct order and reassemble the original messages. Your goal is to design and implement the program for this. This project has three sub-projects, project 2a,2b and 2c, which will be weighted by 30,50 and 20. In project 2a, you will implement two type of linked lists which will be used in project 2b in similar way. In project 2c, you will implement an iterator.

2.2 Purpose

- Reconstruct original messages from partially received packets
- Sort the packets and find missing packets
- Learn about **singly linked list** and **doubly linked list** and compare the performance of the two lists
- Learn about an iterator

2.2.1 Project 2a

Implement linked list structures (Singly & Doubly)

2.2.2 Project 2b

Implement receivers using the implemented linked list in project 2a and compare the running time of receivers using singly and doubly linked list for each

2.2.3 Project 2c

Implement doubly linked list and simple receiver with an iterator

2.3 Ground Rules

Your program must be implemented by using C++ language and runnable on the provided Linux server machines. You are not allows to use any generics of C++. For this project, you can use only the followings.

iostream, stdexcept, string, cstring, ctime, fstream, sstream

2.4 Requirements for Implementing a Network Message Receiver

The arrival of packets will be simulated by an input file with multiple lines, each having the format as follows:

message_number *packet_number* *word*

It is assumed that the word does not contain any blanks. And three values(data) in each line are separated with a tab. For the purposes of the simulation, the word represents the contents of the packet. Output is equally simple. For each message, there is a block of the form

- Message i
word i,1
word i,2
...
word i,n
- End Message i

where word i,j is a packet j of message i. If any packet is missing, i.e., there was no input line for packet j of some message i that had other packets with numbers larger than j, then word i,j should be replaced with warning message as follows:

WARNING: packet j of message i is missing

If there are two packets with the same message and packet number, only the last one to arrive should be used. You do not need to print a warning in this case. Message blocks in the output should be separated by blank lines and sorted in order of increasing message number. Message numbers are not necessarily sequential; some may be missing. At the last line in the output, the program should write the running time of the implemented receiver as follows.

Running Time: [t] s.

In this manner, you should implement two types of the receiver. Functions of the both receivers are the same, but it uses different data structures. One is single receiver which uses a singly linked list structure. Another is double receiver which uses a doubly linked list structure. Both of your programs should be called as single receiver and double receiver for each and executed as follows:

Receiver with singly linked list:

./single_receiver *input_file output_file*

Receiver with doubly linked list:

./double_receiver *input_file output_file*

2.4.1 Example

Suppose the input file (attached 'input_small' file) is

```
1    1    Hi
1    3    one
2    2    every
2    2    only
2    3    one
1    3    two
1    3    three
2    3    two
1    1    Hello
2    3    three
2    2    half
```

The output file (attached 'output_small' file) will be

```
- Message 1
Hello
WARNING: packet 2 of message 1 is missing
three
- End Message 1

- Message 2
WARNING: packet 1 of message 2 is missing
half
three
- End Message 2

Running Time: 0 s.
```

2.4.2 Implementation Hints

You can implement this program any way you like. However, it is recommended that it would be to have a list of message-packets within it. Sorting can be done as you go along, by inserting message-packets in the proper position when they arrive, or at the end.

2.4.3 Comments

Implement a doubly linked list which implements all the basic functions as a list. This simulation is obviously a gross oversimplification of how network traffic really works, but the part about packets not arriving in order is real. Packets sometimes take different routes to avoid congestion or a router along the way may not process them in first-in / first-out order. Or a packet may get lost and need to be resent. The sender typically waits for an acknowledgement that the message has been received and sends the message again if there is no acknowledgement by some fixed delay. Some aspects that are not captured here are :

- The flow of incoming messages never stops

- Messages usually have headers that announce their length and other characteristics
- Packets have lots of other information: some may need to be routed to other destination, some are interpreted as sound or video rather than text, etc.

First, implement a singly linked list, which implements some the basic functions as a list. Specifically,

- single_list(): Constructor of a list
- ~single_list(): Deconstructor of a list their length and other characteristics
- list_get_datak(list_elem *a) : Return k-th data of list element 'a'
- list_get_next(list_elem *a) : Return the next element of 'a' in a list
- list_get_head() : Return the head of a list
- list_insert_front(list_elem *a): Insert element 'a' at the beginning of a list
- list_insert_before(list_elem *a, list_elem *b): Insert element 'b' just before 'a'
- list_insert_after(list_elem *a, list_elem *b): Insert element 'b' just after 'a'
- list_replace(list_elem *a, list_elem *b): Replace element 'a' to 'b'
- list_remove(list_elem *a): Removes an element from its list and deconstruct the element
- list_empty(): True if a list is empty, false otherwise.

For singly linked list implementation, we provide template codes (*single_list.h*, *single_list.cpp*) with more detailed explanation. In single list.h, the basic structure of singly linked list is already defined. You should not change *single_list.h*, but refer to the header file to implement functions in *single_list.cpp*. In *single_list.cpp*, we provide template of functions and you should fill out the functions. In other words, you should implement singly linked list by modifying only *single_list.cpp*. Second, implement a doubly linked list, which implements some the basic functions as a list. Specifically,

- double_list(): Constructor of a list
- ~double_list(): Deconstructor of a list
- d_list_get_datak(d_list_elem *a) : Return k-th data of list element 'a'
- d_list_next(d_list_elem *a) : Return the next element of 'a' in a list
- d_list_prev(d_list_elem *a) : Return the previous element of 'a' in a list
- d_list_head() : Return the head of a list.
- d_list_tail() : Return the tail of a list
- d_list_front() : Return the front(first element in a list) of a list
- d_list_back() : Return the back(last element in a list) of a list
- d_list_insert_front(d_list_elem *a): Insert element 'a' at the beginning of a list.
- d_list_insert_back(d_list_elem *a): Insert element 'a' at the end of a list

- d_list_insert_before(d_list_elem *a, d_list_elem *b): Insert element 'b' just before 'a'
- d_list_insert_after(d_list_elem *a, d_list_elem *b): Insert element 'b' just after 'a'
- d_list_replace(d_list_elem *a, d_list_elem *b): Replace element 'a' to 'b'
- d_list_remove(d_list_elem *a): Removes an element from its list and deconstruct the element.
- d_list_empty(): True if a list is empty, false otherwise.

For doubly linked list implementation, we provide template codes (*double_list.h*, *double_list.cpp*) with more detailed explanation. In *double_list.h*, the basic structure of doubly linked list is already defined. You should not change double list.h, but refer to the header file to implement functions in *double_list.cpp*. In *double_list.cpp*, we provide template of functions and you should fill out the functions. In other words, you should implement doubly linked list by modifying only *double_list.cpp*.

Please start from the provided code and refer to the book. Note that there could be some differences between the given code in our project and the code in the book. You should strictly follow the format(input, output, name of functions) of the provided code in our project. Also, defining new function is not allowed. We will not care about any errors occurring because of the difference of the format when we test your codes.

Note that Project 2a is a stepping stone to a Project 2b. Submit followings in the form of compressed file(tar.gz) with the title [EE205project2a]20XXXXXX.tar.gz on the KLMS web page.

- *single_list.cpp*, *double_list.cpp* : Implementation of singly linked list and doubly linked list.
- *readme* : Specific consideration for your code or anything ambiguous in interpreting this project

2.4.4 Project 2b

Complete two Network Message Receivers which are implemented by using singly linked list and doubly linked list. You should implement both type of receiver(single receiver, double receiver) which receive unsorted packets from an input file and output the sequence of every packets following the rule specified in the 2.4. As mentioned in the 2.4, contents that the output file should contain the following:

- Sorted contents of messages in input file
- Running time of the receiver

To check the time taken, use difftime() function in **ctime**. You should write the result of the time comparison between the two receivers when input file is input_large in a *readme* file with the explanation about the program flow of your receiver.

The receivers should take two arguments, the first argument is the name of input file and the second argument is the name of output file. You can see the execution example of receivers in the attached file and 2.4.1

For receiver implementation, we provide template codes (*single_receiver.cpp*, *double_receiver.cpp*). They include *single_list.h*, *double_list.h* for each, so that you can use the functions of linked list implemented in project 2a. You should not change header files and include more header files in *single/double_receiver.cpp*. In other words, you should implement receivers by modifying only the inside of main function in *single_receiver.cpp*, *double_receiver.cpp*.

You should strictly follow the rule stated in the section 2.4. Also, defining new function is not allowed. We will not care about any errors caused from the violation of the rules.

Submit followings in the form of compressed file(tar.gz) with the title [EE205project2b]20XXXXXX.tar.gz on the KLMS web page.

- *single_list.cpp*, *double_list.cpp*, *single_receiver.cpp*, *double_receiver.cpp* : Implementation of packet receivers using singly/doubly linked list data structure.
- *readme*: Explanation of your program flow of each receiver code, time comparison result between the receiver programs using singly/doubly linked list, and specific consideration for your code or anything ambiguous in interpreting this project.

2.4.5 Project 2c

Goal of the project 2c is to implement iterator. Unlike project 2a and 2b, header file can be modified and only doubly linked list implementation is required. The files you need to modify are in the '2c' folder.

The objective of the main function is much simpler than project 2b, you only need to print one line whether the 'Message 4' exists or not. This task does not need to use a list or any data structures, but it is a simple example for implementing an iterator. The basic code for the main function in *double_receiver_iter.cpp* is given. Minor changes do not matter, but make it based on that code. We use additional library **algorithm** in this task to use find() function. find() returns an iterator to the first element in the range [first, last) that equals to val or last if no such element is found. For more information, please visit here,

http://www.cplusplus.com/reference/algorithm/find/

Because we use a custom structure, we need to implement an iterator class to use the find function. When referring to the nested class Iterator, we need to apply the scope resolution operator, as in double_list::Iterator, so the compiler knows that we are referring to the iterator type associated with double_list. In the iterator class, there are equality/inequality operators ("==" and "!="), increment/decrement operators ("++" and "–") and reference to the element. Since we are only interested message (not packet) in this task, we refer to message number (data1) here. It is described in more detail in chapter 6.2.3 of the textbook.

Submit followings in the form of compressed file(tar.gz) with the title [EE205project2c]20XXXXXX.tar.gz on the KLMS web page.

- *double_list_iter.cpp*, *double_list_iter.h* *double_receiver_iter.cpp* : Implementation of an iterator and simple packet receiver using doubly linked list data structure.
- *readme*: Explanation of your program flow of your codes, and specific consideration for your code or anything ambiguous in interpreting this project.

2.5 Evaluation

2.5.1 Project 2a

TA will evaluate your implementation by testing TAs packet receiver program which utilize the functions of singly/doubly linked lists you implemented. The TAs packet receiver will run in a similar way to the receiver described for the project 2b. It will take an input file which contains unsorted messages and reorganize

the messages using your linked list implementation. We will check whether TAs packet receiver outputs correctly for several testing input files. Since the TAs packet receiver call only the functions listed in single list.h double list.h to reorganize messages, the correctness of output of the TAs packet receiver is depending on the correctness of your linked list implementation.

We will compare (the output file of TAs packet receiver compiled with your list implementation) with (the output file of TAs packet receiver compiled with TAs correct list implementation). We will check each functionality of the list functions using several input files which incur various corner cases. The example of test process is described below. Example of input file is provided with the template codes of the project. Note that TA will will use more various input files to test all the corner cases.

g++ -o TA_you_single_receiver TA_single_receiver your_single_list.cpp

g++ -o TA_TA_single_receiver TA_single_receiver TAs_single_list.cpp

./TA_you_single_receiver input_file you_output_file

./TA_TA_single_receiver input_file TA_output_file

diff you_output_file TA_output_file

2.5.2 Project 2b

Your receiver program will take an input file which contains unsorted messages and should output reorganized messages using your linked list implementation. We will check whether your packet receiver outputs correctly for several testing input files. The correctness will be checked by comparing the two output files from your receiver and TAs receiver for the same input file. There should be no difference between the two output files except the execution time part, if you implemented your receiver to run properly as described in the section 2.4. So, you should strictly follow the output format when you implement the function of writing output file.

We will use ‘diff function to check the difference of the output files and will not take any claim for errors caused from the violation of writing format. We will check each functionality of your packet receiver using several input files which incur various corner cases. The example of test process is described below. Example of input file is provided with the template codes of the project. Note that TA will will use more various input files to test all the corner cases

g++ -o you_you_single_receiver your_single_receiver your_single_list.cpp

g++ -o TA_TA_single_receiver TA_single_receiver TA_single_list.cpp

./you_you_single_receiver input_file you_output_file

./TA_TA_single_receiver input_file TA_output_file

diff you_output_file TA_output_file

2.5.3 Project 2c

Your receiver program will take an input file which contains unsorted messages and should output there exists ’Message 4’ or not using your linked list implementation. We will check whether your packet receiver outputs correctly for several testing input files. The correctness will be checked by comparing the two output files from your receiver and TAs receiver for the same input file. In addition, TA will check the code to see it has implemented properly in the project 2c.

g++ -o you_you_double_receiver_iter your_double_receiver_iter your_double_list_iter.cpp

g++ -o TA_TA_double_receiver_iter TA_double_receiver_iter TA_double_list_iter.cpp

./you_you_double_receiver_iter input_file you_output_file

./TA_TA_double_receiver_iter input_file TA_output_file

diff you_output_file TA_output_file

2.6 Grading

2.6.1 Project 2a: 30%

- Correct implementation and result on the example files for singly linked list: 5%
- Correct implementation and result on the example files for doubly linked list: 10%
- Correct implementation and result on edge cases for singly linked list: 5%
- Correct implementation and result on edge cases for doubly linked list: 10%

2.6.2 Project 2b: 50%

- Correct implementation and result on the example files for singly linked list: 5%
- Correct implementation and result on the example files for doubly linked list: 10%
- Correct implementation and result on edge cases for singly linked list: 5%
- Correct implementation and result on edge cases for doubly linked list: 10%
- Correct explanation about functions, program flow: 10%
- Time comparison between two list structures : 10%

2.6.3 Project 2c: 20%

- Correct implementation and result on the example files: 10%
- Correct explanation about functions, program flow: 10%

readme will not be considered when grading in project 2a. But, if there is any unclear things, your detailed explanation would help TA to understand your code much better and to grade your project 2a properly. And, please note that the contents of *readme* in project 2b, 2c will be significantly considered.

In project 2c, Writing the *readme* file properly helps in scoring. Since the iterator and the list class functions are not functionally different, it is not possible to score by output alone. Therefore, the TA will look at the code in detail.

You don't need to consider the cases that the input command does not follow the exact form. (Ex. You can assume that list_getdata always get an element exists in the list. That means, you don't need to handle the case that the argument *elem is not an element of a list.)

For the undefined case, you don't need to consider it as **stderr**. But if there is something to return for a function, you should return NULL when undefined case occurs.

2.6.4 Due Date & Delay Penalty

Due date: Wed 24th Oct, 23:59:59

Final due date: Sun 28th Oct, 23:59:59 with a 25% per day penalty

After final due, you will get 0 score.