EE 205 Data Structures and Algorithms **Fall 2017**

Project 2: Receiving Messages from a Packet Network

Lecturer: Yung Yi *TA: Jiin Woo*

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 two sub-projects, project #2a and project #2b, which will be weighted by 40 and 60, respectively. In project #2a, you will implement two type of linked lists which will be used in project #2b in similar way.

2.2 Purpose

Learn about singly linked list and doubly linked list and compare the performance of the two lists.

2.3 Ground rules

Your program must be implemented by using C++ language and runnable on the provided Linux server machines. You are not allowed 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 value(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_reciver' which uses a singly linked list structure. Another is 'double_reciver' 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.txt' file) is

```
3       2       morning
17       1        See
10       3        you
17       2        forget
10       1        How
17       3        later
3       1       Good
17       2        you
10       4        doing
```

The output will be

```
- Message 3
Good
morning
- End Message 3

- Message 10
How
WARNING: packet 2 of message 10 is missing
you
doing
- End Message 10

- Message 17
See
you
later
- End Message 17

Running Time: 6 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 resends the message if there is no acknowledgement by some fixed delay. Some aspects that are not captured here are :

- o The flow of incoming messages never stops
- o Messages usually have headers that announce their length and other characteristics
- o 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.

2.4.4 Project #2a

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

- o single_list(): Constructor of a list
- o ~single_list(): Deconstructor of a list
- o list_get_datak(list_elem *a) : Return k-th data of list element 'a'.
- o list_get_next(list_elem *a) : Return the next element of 'a' in a list.
- o list_get_head() : Return the head of a list.
- o list_insert_front(list_elem *a): Insert element 'a' at the beginning of a list.
- o list_insert_before(list_elem *a, list_elem *b): Insert element 'b' just before 'a'.
- o list_insert_after(list_elem *a, list_elem *b): Insert element 'b' just after 'a'.
- o list_replace(list_elem *a, list_elem *b): Replace element 'a' to 'b'.
- o list_remove(list_elem *a): Removes an element from its list and returns the element that followed it.
- o 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 shoud 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,

- o double_list(): Constructor of a list
- o ~double_list(): Deconstructor of a list
- o d_list_get_datak(d_list_elem *a) : Return k-th data of list element 'a'.
- o d_list_next(d_list_elem *a) : Return the next element of 'a' in a list.
- o 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 returns the element that followed it.
- 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 shoud 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.

There could be some cases that list has some same elements (but in project #2b it does not occur). If there are duplicated elements in the list, insert_before, insert_after, functions consider the first element in the list, but replace function considers all duplicated elements in the list.

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 [EE205 project2a]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.

2.4.5 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 section 2.4.1. As menetioned in the section 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 section 2.4.

For receiver implementation, we provide template codes (*single_receiver.cpp, double_receiver.cpp*). They inlucde *single_list.h, double_list.h* for each, so that you can use the functions of linked list implemented in project #2a. You shoud 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 [EE205 project2b]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 comarison result between the receiver programs using singly/doubly linked list.
  Specific 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 TA's packet receiver program which utilize the functions of singly/doubly linked lists you implemented. The TA's 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 TA's packet receiver outputs correctly for several testing input files. Since the TA's packet receiver call only the functions listed in *single_list.h double_list.h* to reorganize messages, the correctness of output of the TA's packet receiver is depending on the correctness of your linked list implementation. We will compare (the output file of TA's packet receiver compiled with your list implementation) with (the output file of TA's packet receiver compiled with TA's 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 descriabed 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

As described in the section 2.4.5, 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 TA's 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 descriabed 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 TAs_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
```

2.6 Grading

Project 2a: 40%

- Correct implementation and result on edge cases for singly linked list: 20%
- Correct implementation and result on edge cases for doubly linked list: 20%

Project 2b: 60%

- Compile and execution on the example input file without error : 5%
- Correct explanation about functions, program flow: 5%
- Correct result on edge cases & large test input data: 40%
- Time comparison between two list structures : 10%

2.7 Delay penality

You will get 80% of your project2 score until 24 hours after the due date, and 0% after more than 24 hours.