**EE6094 CAD for VLSI Design**

**Programming Assignment 2: Scheduling**

Student ID: 108501023

Student Name: 李品賢

**Compile, Execute and Verification**

1. Pull the source code, i.e., *108501023\_PA2.cpp*, *Makefile*, *testcase1*, *testcase2*, *testcase3* and *checker* into the workstation folder.

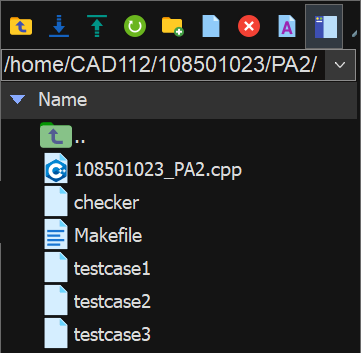


Fig. 1

1. Use *Makefile* as a trigger point to run the *108501023\_PA2.cpp* program, and then the output *testcase1.out* / *testcase2.out* / *testcase3.out* are generated.
   * make all
   * make run Testcase=testcase1
   * make run Testcase=testcase2
   * make run Testcase=testcase3
   * make clean

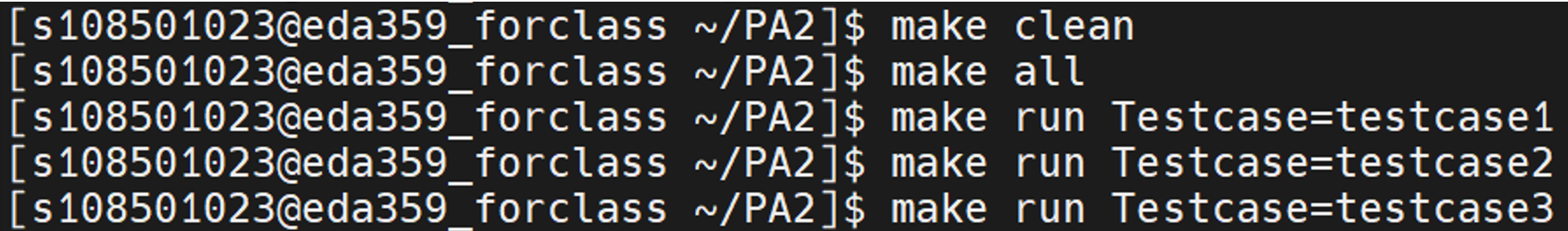


Fig. 2

1. Use checker to check whether output files fits the standard output format. The screenshots of the command are included in completion part.
   * ./checker testcase1 testcase1.out
   * ./checker testcase2 testcase2.out
   * ./checker testcase3 testcase3.out

**Completion**

All three testcases are successfully passed the checker, the screen shows three cute “Nyan Cat”. The following three figure (Fig. 1, Fig. 2, Fig. 3) are the results.

|  |  |  |  |
| --- | --- | --- | --- |
|  | testcase1 | testcase2 | testcase3 |
| Completion | O | O | O |
| Hardware | 4 | 19 | 1393 |

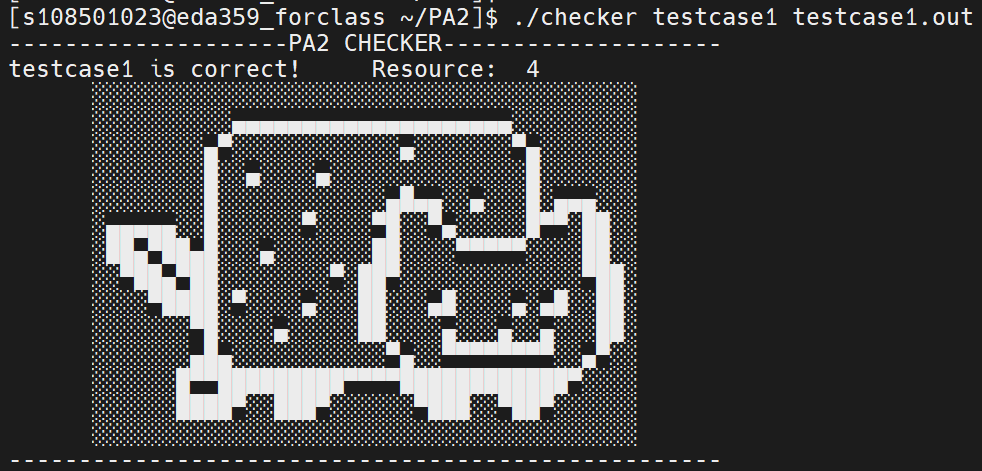


Fig. 3

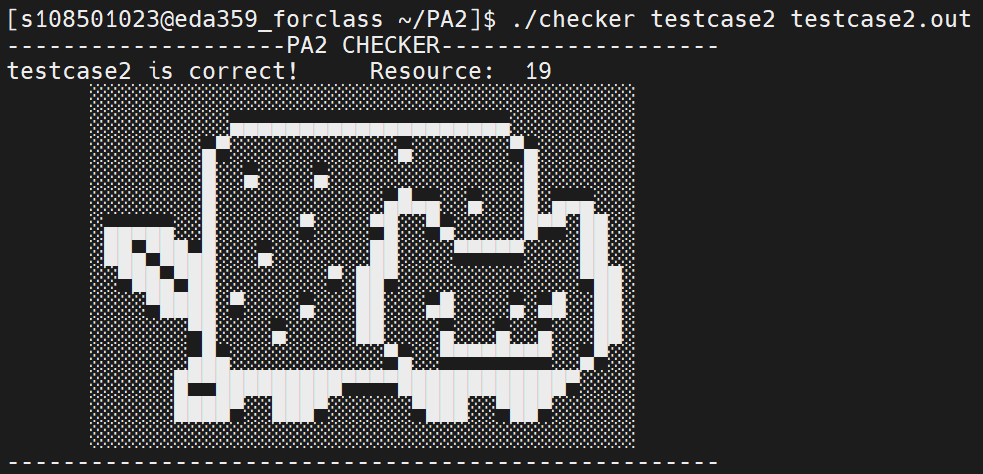


Fig. 4

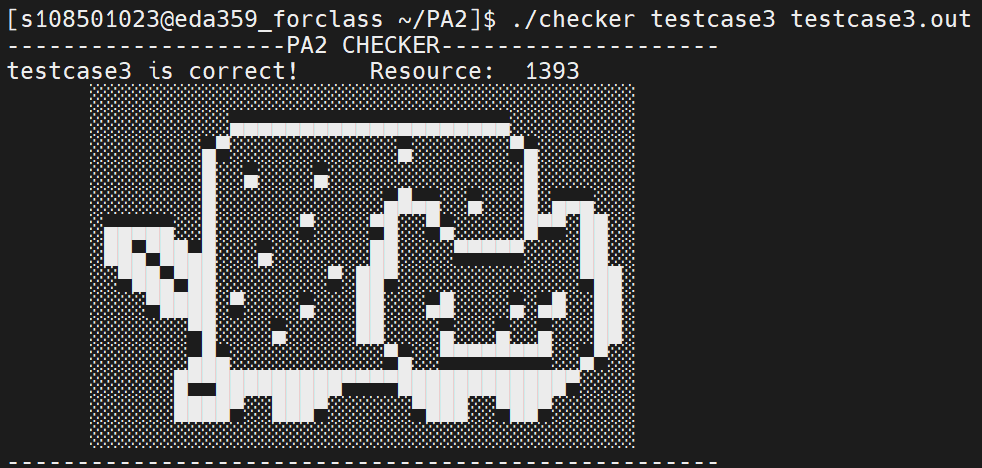


Fig. 5

**Data Structure and Algorithm**

Data structure:

I use 2-D linked-list array (left-side of Fig. 4) to represent a graph (right-side of Fig. 4). The grandparent relationship between each node can be included in this representation. The greatest benefit of this structure is that it takes fewer time to implement topology sort either from the top or the bottom, both of which are the algorithms included in this program.

For the starter, this structure actually represents a matrix. (Fig. 5) Each row of the matrix represents the out-degree of the vertex, e.g., matrix[0][1] is equal to 1, which means that vertex 0 points to vertex 1. In contrast, each column represents the in-degree of the vertex.

Headnodes and nodes are used. The x-axis headnodes record the successor relationship while the y-axis headnodes record the predecessor relationship. Nodes record edge information and point to another node by the according predecessor or successor relationship.

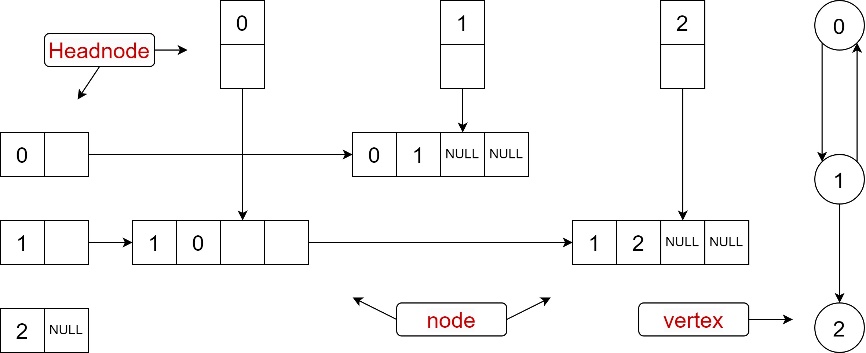


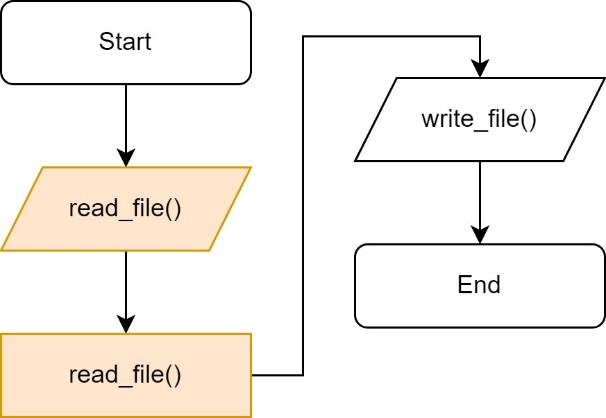
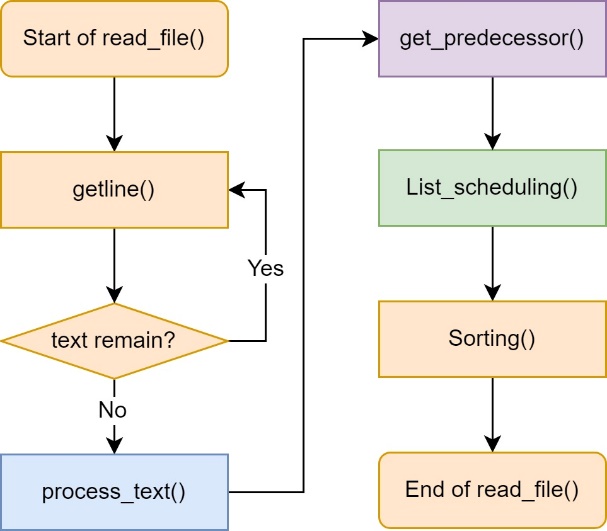
Fig. 6

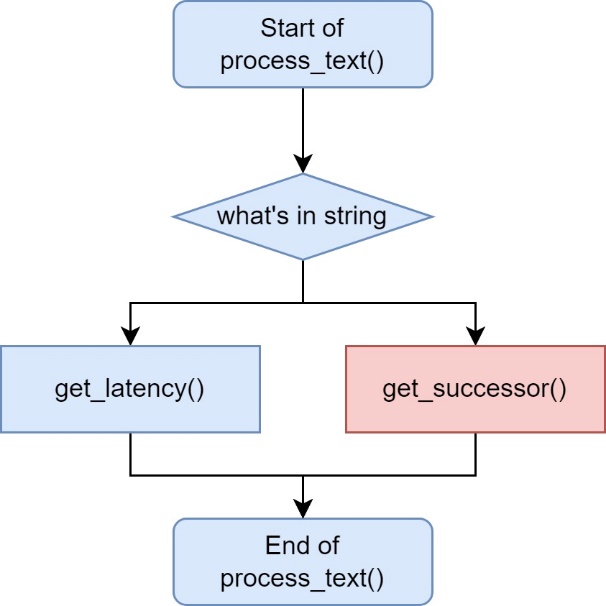
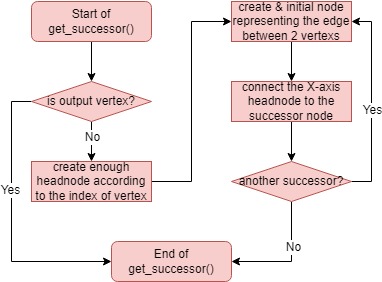
Fig. 7

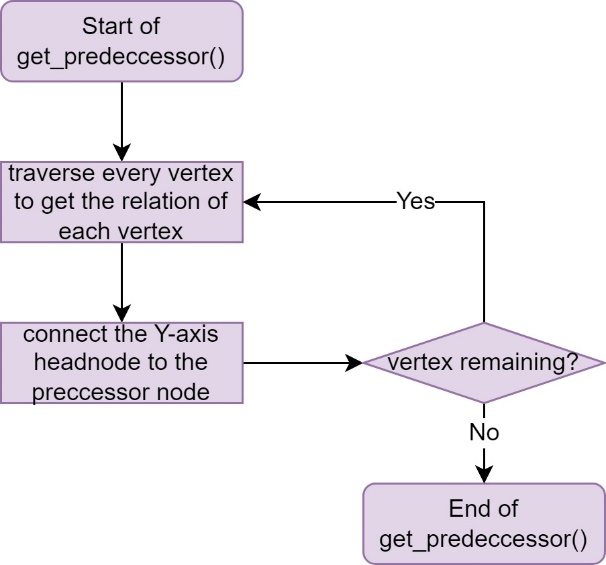
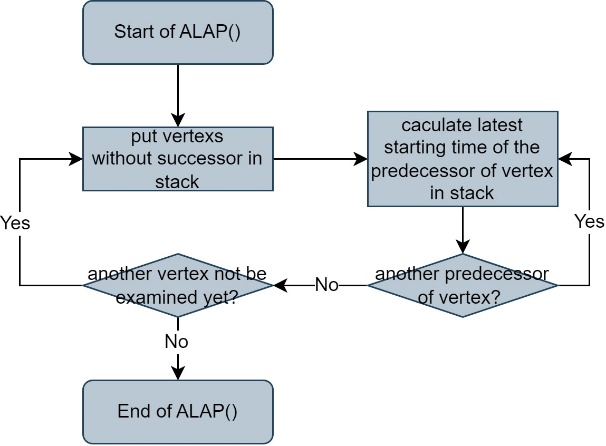
Algorithm:

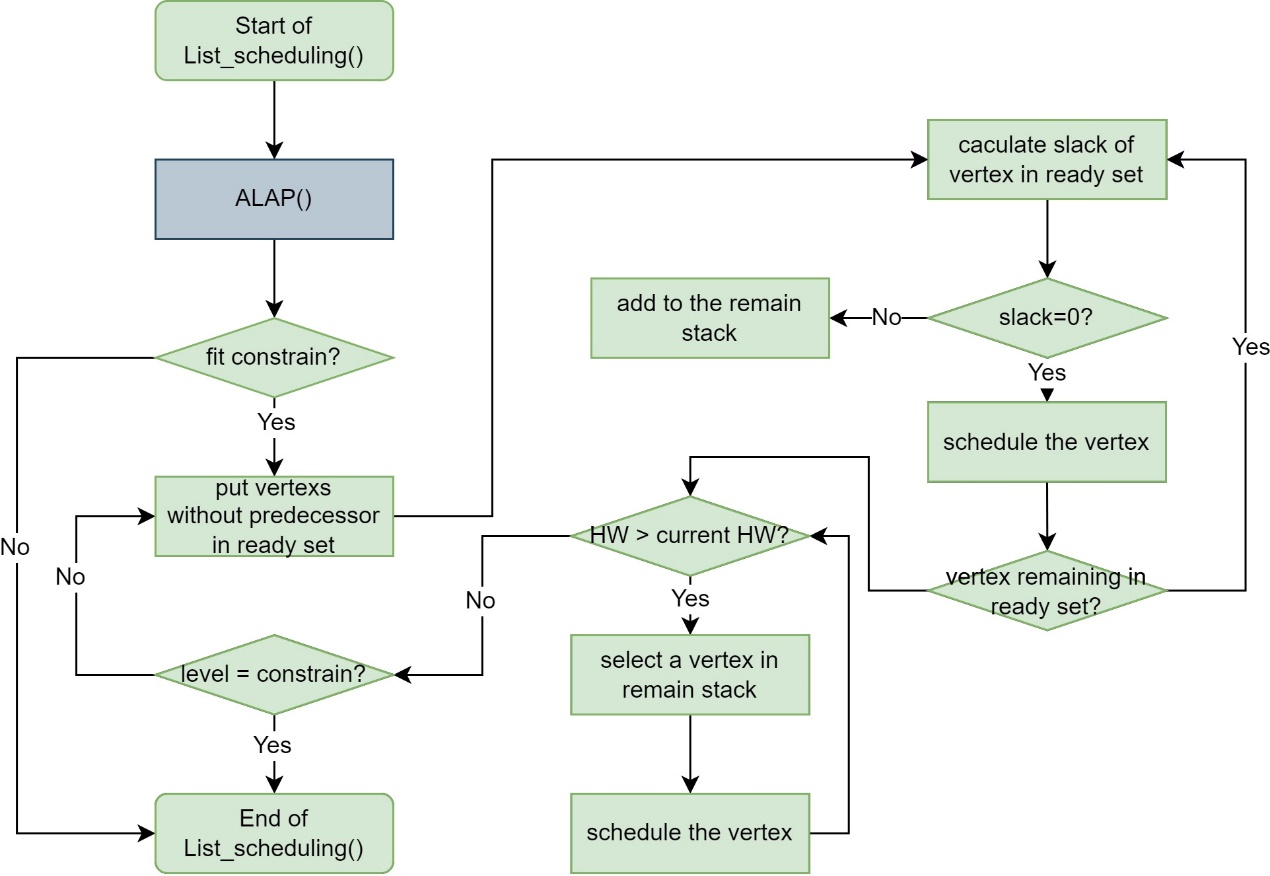
I use list scheduling algorithm to complete this assignment. The other algorithm inside the list scheduling algorithm is ALAP algorithm which use the concept of topology sort. After ALAP algorithm executed, the algorithm of scheduling part also uses the concept of topology sort. The difference between two scenarios is mentioned above, which is that ALAP algorithm apply topology order staring from bottom, while the other use that starting from the top.

**Flow Chart**

** **

** **

** **

****

**Source Code Explanation**

Data structure:

1. struct node

The *struct node* structure (Fig. 6) stores the edge information (*int suc, pre*). The variable *int pre* represents the vertex closer to the top, while the variable *int suc* represents the vertex closer to the bottom. The variable *int cycle* stores execution time of the vertex *“pre”.* Also, because it is the node in 2-D linked-list array, potiners to the successor and predecessor are included (*struct node \*sucPtr, \*prePtr*).

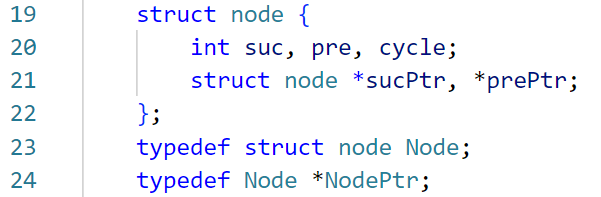


Fig. 8

1. struct vertexInfo

The *struct vertexInfo* structure (Fig. 7) stores the vertex information including latest starting time (*int latest*) for ALAP algorithm, successor counts (*int sucCount*), predecessor counts (*int preCount*), operation types (*char oper*) and valid bits (*bool valid*) for recognizing whether this vertex is included in input file and for representing the scheduled vertex.

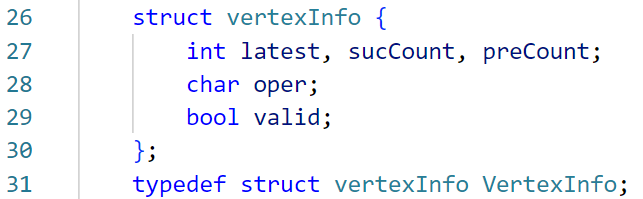


Fig. 9

Variable:

1. int latency

store the latency constrain

1. int ADDCount, MULCount, vertexNum

store the total adder count, multiplier count and vertex count.

1. string line

catch the text of input file line by line

1. vector<NodePtr> sucHead, preHead

headnodes for the 2-D linked-list array. *sucHead* is a x-axis header for row, while y-axis *preHead* is for column.

1. vector<VectorInfo> vertex

use a vector as a container to store the vertex information

1. vector< vector<int>> output

use a 2-D vector to store the scheduled vertex number in each level

1. vector< vector<int>> MULstate

record the index of specific vertexes with type ‘\*’ and executing cycle of it.

Function:

1. void read\_file()

read input file.

1. void write\_file()

write output file.

1. void process\_text()

process text including latency and vertex information from input line.

1. int str\_to\_int()

turn the string to the integer.

1. void get\_successor()

construct the 2-D structure with horizontal direction according to the incoming vertex information.

1. void get\_ predecessor ()

construct the 2-D structure with vertical direction according to information of current 1-D table.

1. int get\_latency()

receive the latency constrain.

1. void extend\_vertex\_count()

create enough headnode according to the incoming vertex number.

1. void create\_node(int, int)

create a node in 2-D structure. Arguments of function are both end of edge.

1. bool ALAP()

implement ALAP algorithm

1. void List\_Scheduling()

implement list scheduling algorithm.

1. void schedule\_vertex(int, int, char, vector<int> &)

If the condition is meet, then schedule the specific vertex. Arguments of function are index of vertex, level of time step, operation type and vector MULstate.

1. decrement\_preCount(int)

called at the end of execution cycle of specific vertex, i.e., first cycle of ‘i’ vertex, first cycle of ‘+’ vertex and the third cycle of ‘\*’ vertex.

1. void sorting()

sort the vertex number in ascending order in each time step.

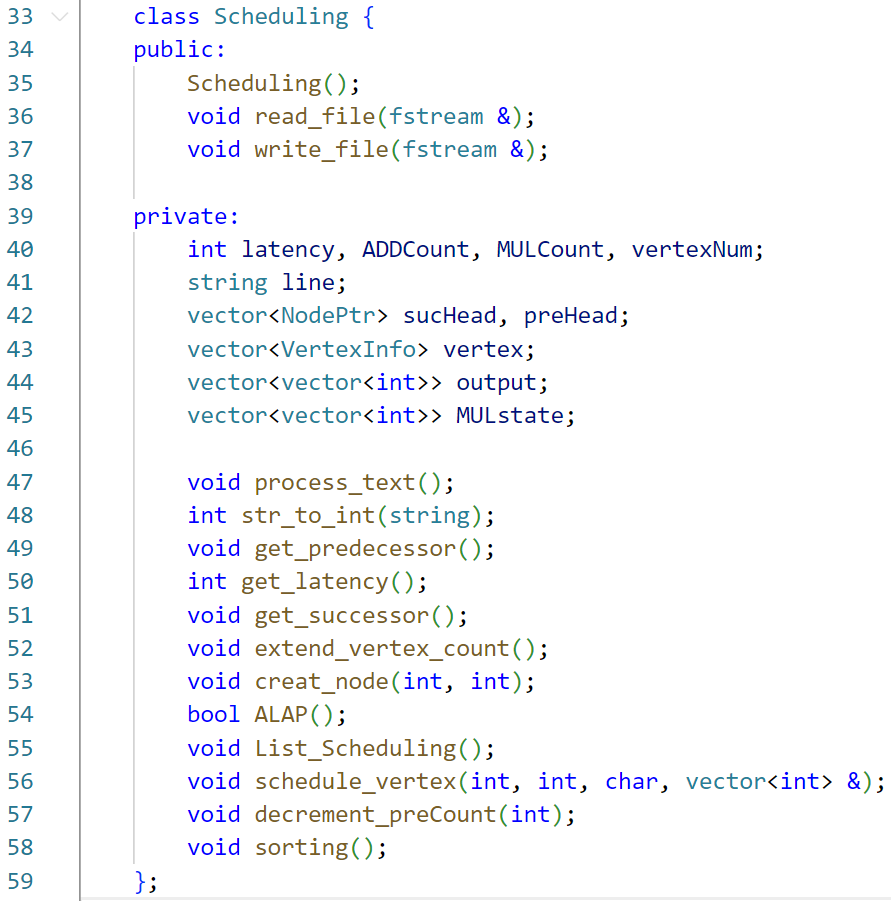


Fig. 10

Code:

In the code explanation, I will explain the most important part in my source code, such as main(), *process\_text(), get\_predecessor(), get\_latency(), get\_successor(), ALAP(), List\_Scheduling(), schedule\_vertex()* and *decrement\_preCount().*

1. int main()

The function of main function is to trigger the entire program, so it only contains the function of read input file and write output file.

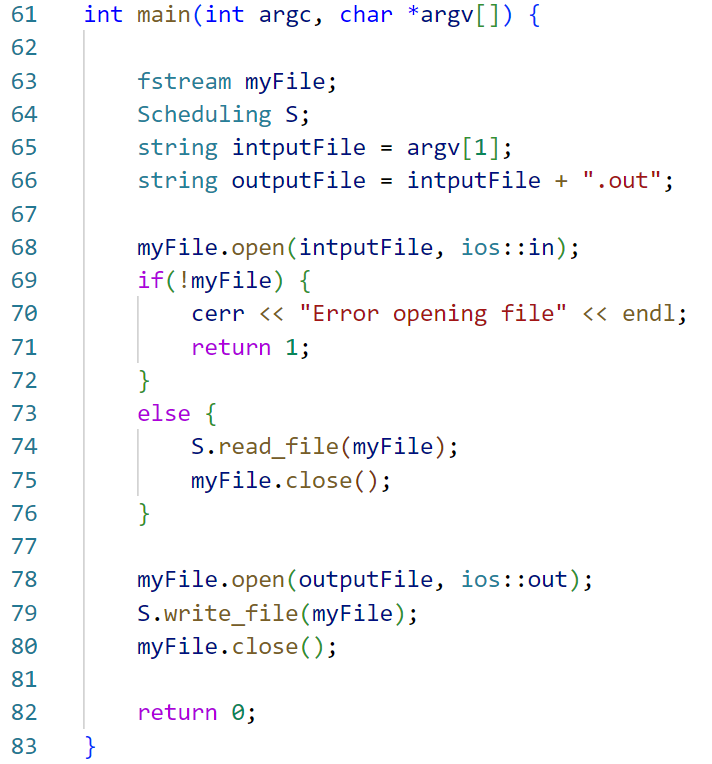


Fig. 11

1. void read\_file()

There are four step in this function. For the starter, it catchs the line string of input file through *getline()*, and call *process\_text()* to get either the latency constrain or the vertex information so as to construct the 1-D linked-list array.

Then, it calls *get\_predecessor()* to reconstruct 1-D linked-list array to one with 2-D.

Furthermore, it implements the scheduling by calling *List\_Scheduling()*.

Finally, sorting the output the vertex number in ascending order in each time step is the last step by calling *sorting()*.

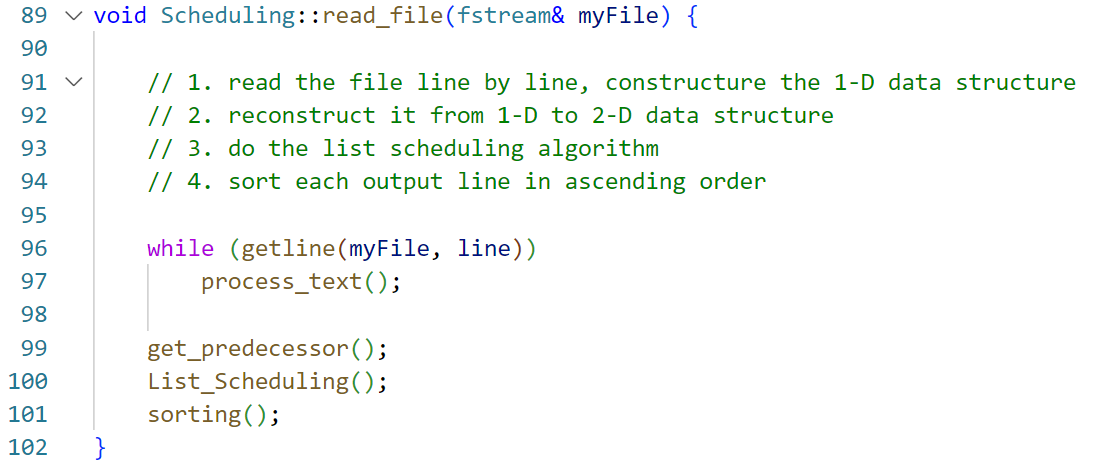


Fig. 12

1. void process\_text()

If the string “Latency constrain” is detected within input line string, it will receive the latency by calling *get\_latency()*. I will not explain this function due to its straightforward implementation.

Also, if the starting character of input line string is a integer number, it calls *get\_succerror()*, which construct the 1-D array by the vertex information from the input line string.

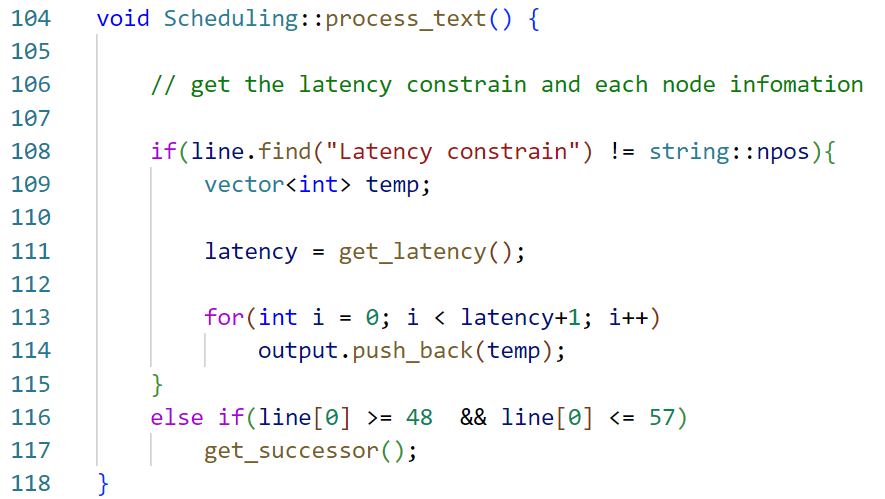


Fig. 13

1. void get\_successor()

If the incoming vertex is not output vertex and there is any successor according to the line string, then it enters the while loop (code 158).

The loop does two main things, one is creating enough headnodes according to the incoming vertex number (code 163-165), another one is creating the node in linked-list array, using x-axis headnode point to it and increasing the successor count of the incoming vertex by 1 (code 167-168).

After end of loop, it sets the valid bit of incoming vertex and stores the operation type of it (code 171-172).

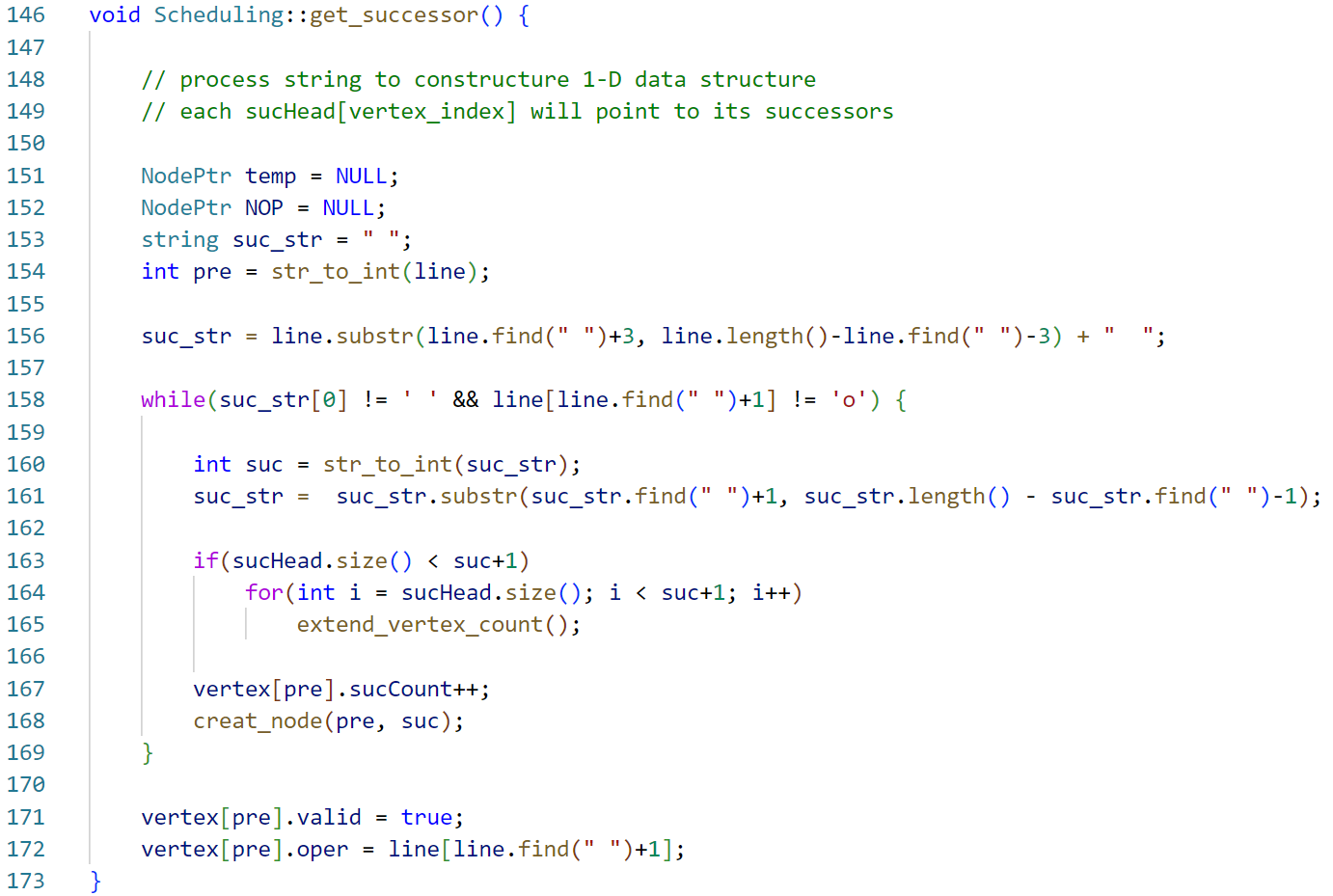


Fig. 14

1. void get\_predecessor()

The for loop (code 214) examines every x-axis headnode to receive the edge information.

Then, the while loop (code 218) does three things, one is increasing the predecessor count of the vertex closer to the bottom by 1. Another is assigning the delay time in each node (code 219-226). The delay time stored in each node represents the execution time of the vertex closer to the top (“*pre*”) in edge relation. The other is connecting nodes in 1-D table to y-axis headnode (code 228-230)

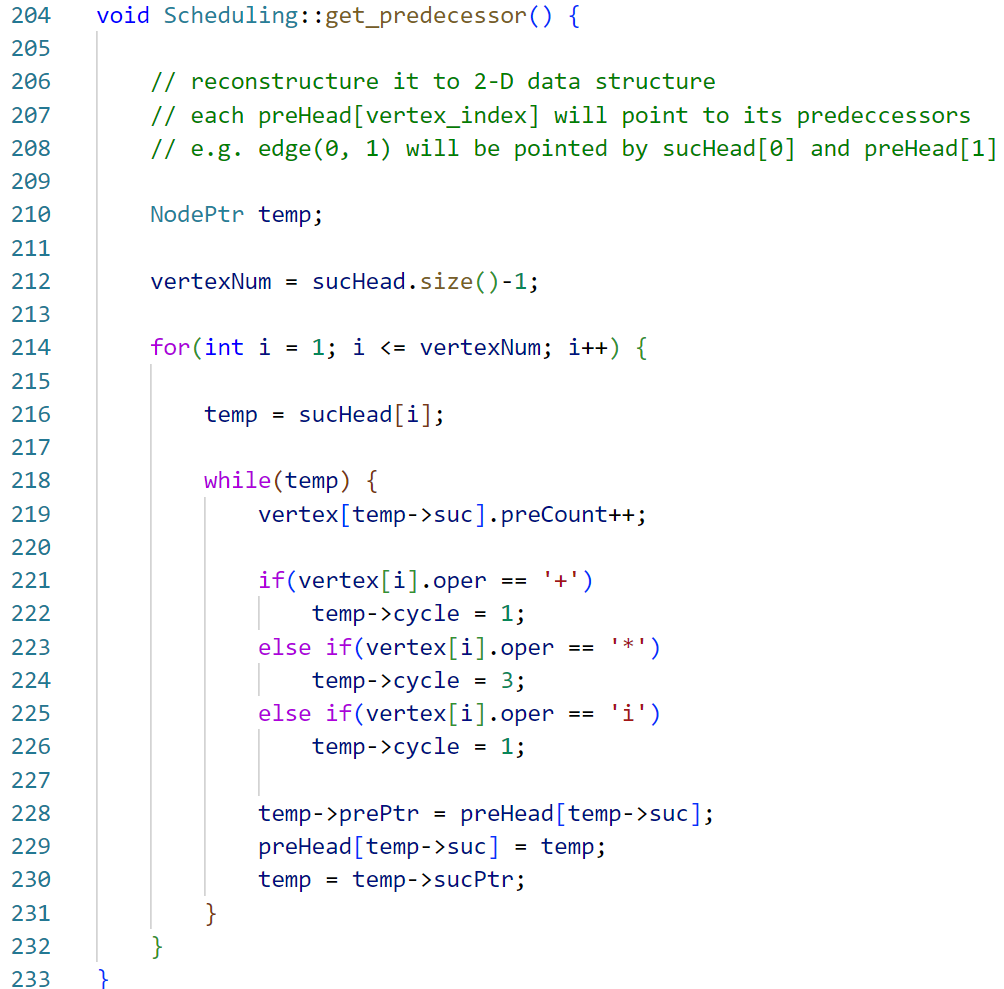


Fig. 15

1. bool ALAP()

The first for loop (code 242) searches all vertex to find the valid one with 0 successor count, and put them into the stack.

The second for loop (code 246) starts calculating latest starting time of each vertex with 0 successor count.

The for loop (code 250) inside second loop examines every predecessor of the vertex. First, it decrements the successor count of predecessor by 1 (code 252), and then put it into stack if its successor count is 0 (code 254-255). Because this is a ALAP algorithm, the predecessor chooses the minimum value of difference between it and its successor (code 257-259). By the way, the initial value of latest starting time of each node is latency+1. Notice that if there is a latest starting time of predecessor is less than 0, *ALAP()* return false, which means the scenario of this input file is impossible to solve in this latency constrain (code 261-262).

If it finally reach the end of *ALAP()*, it return true (code 266).

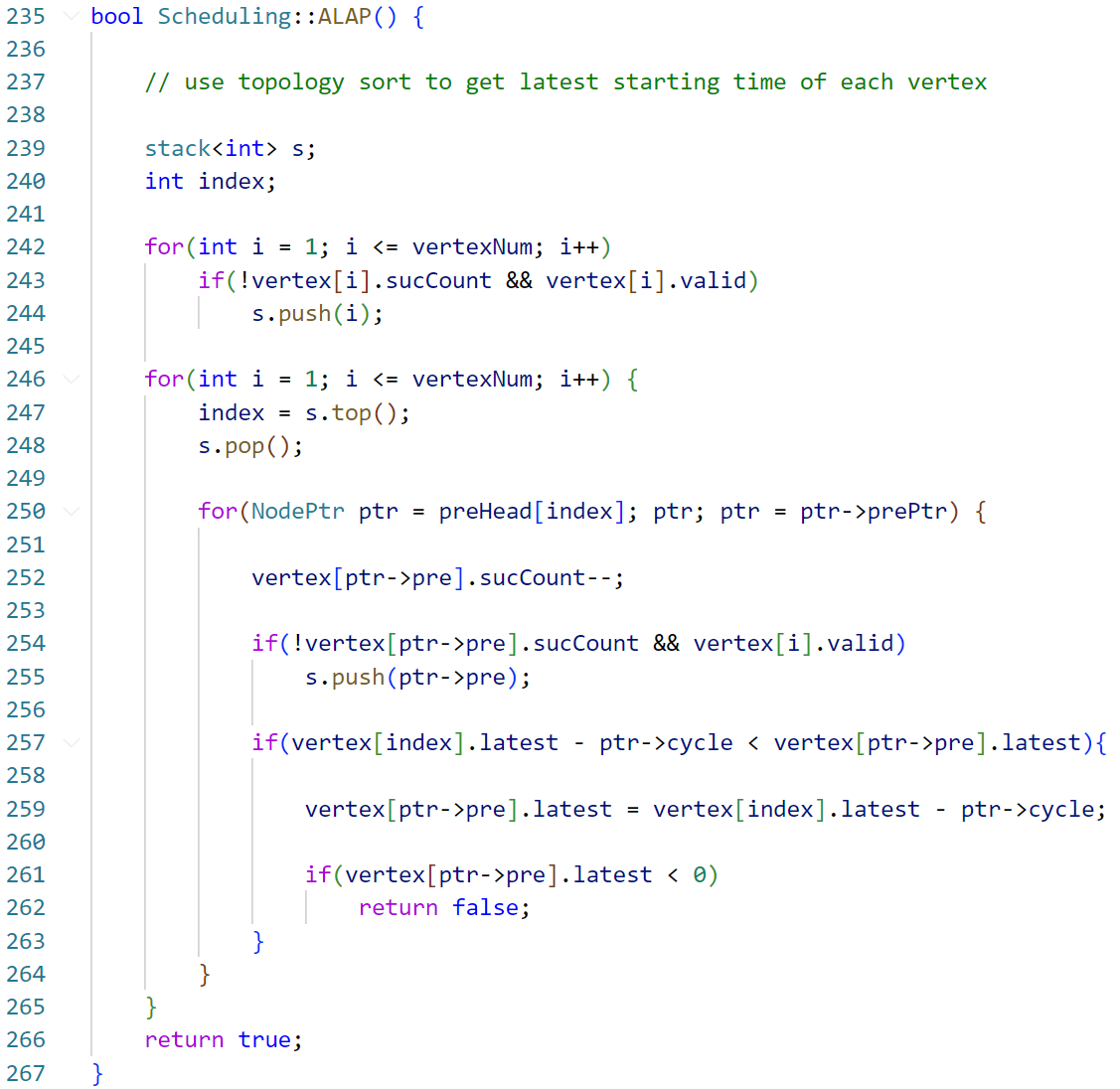


Fig. 16

1. void List\_Scheduling()

First, it calls *ALAP()*. If *ALAP()* returns false, then this function is unnecessary to execute, and prints the waring statement.

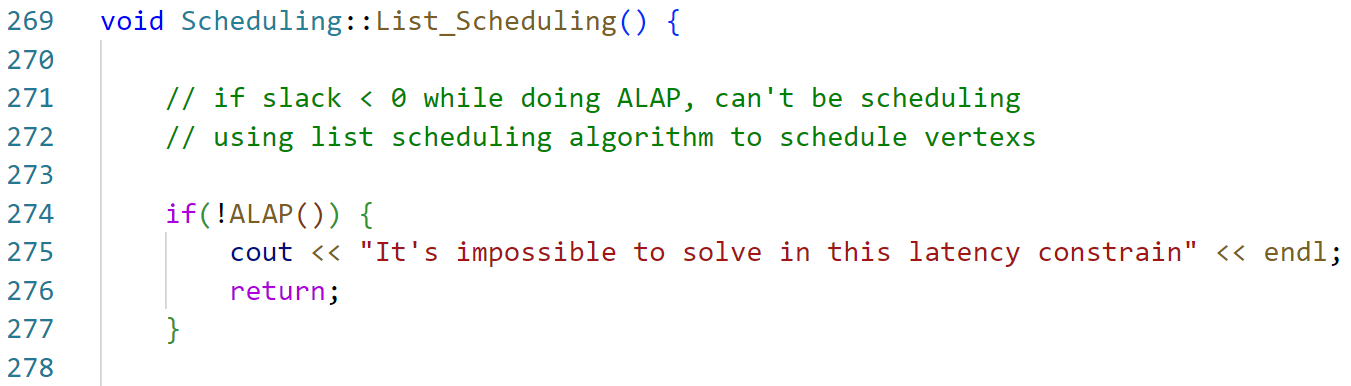


Fig. 17

If *ALAP()* return true, then it starts to schedule every vertex in each time step.

The first for loop (code 289) examines every time level until level is equal to latency. The level is used to calculate the slack later.

The for loop (code 291) inside first for loop searches all vertex to find the valid one with 0 predecessor count, and put them into the “ready queue”.

The for loop (code 297) examines every vertex in “ready queue” and calculates the slack which is the difference of latest starting time of vertex and level (code 302). Afterwards, if it’s input vertex, there is nothing to do, it calls *decrement\_preCount()* directly (to decrease all successor’s *preCount*) (code 304-305). Continued, if it’s slack is 0, it calls *schedule\_vertex()* accordingly to do some operation, which will be explained in the text later (code 306-311). Otherwise, the vertex should be push into “remain stack” (code 312-313).

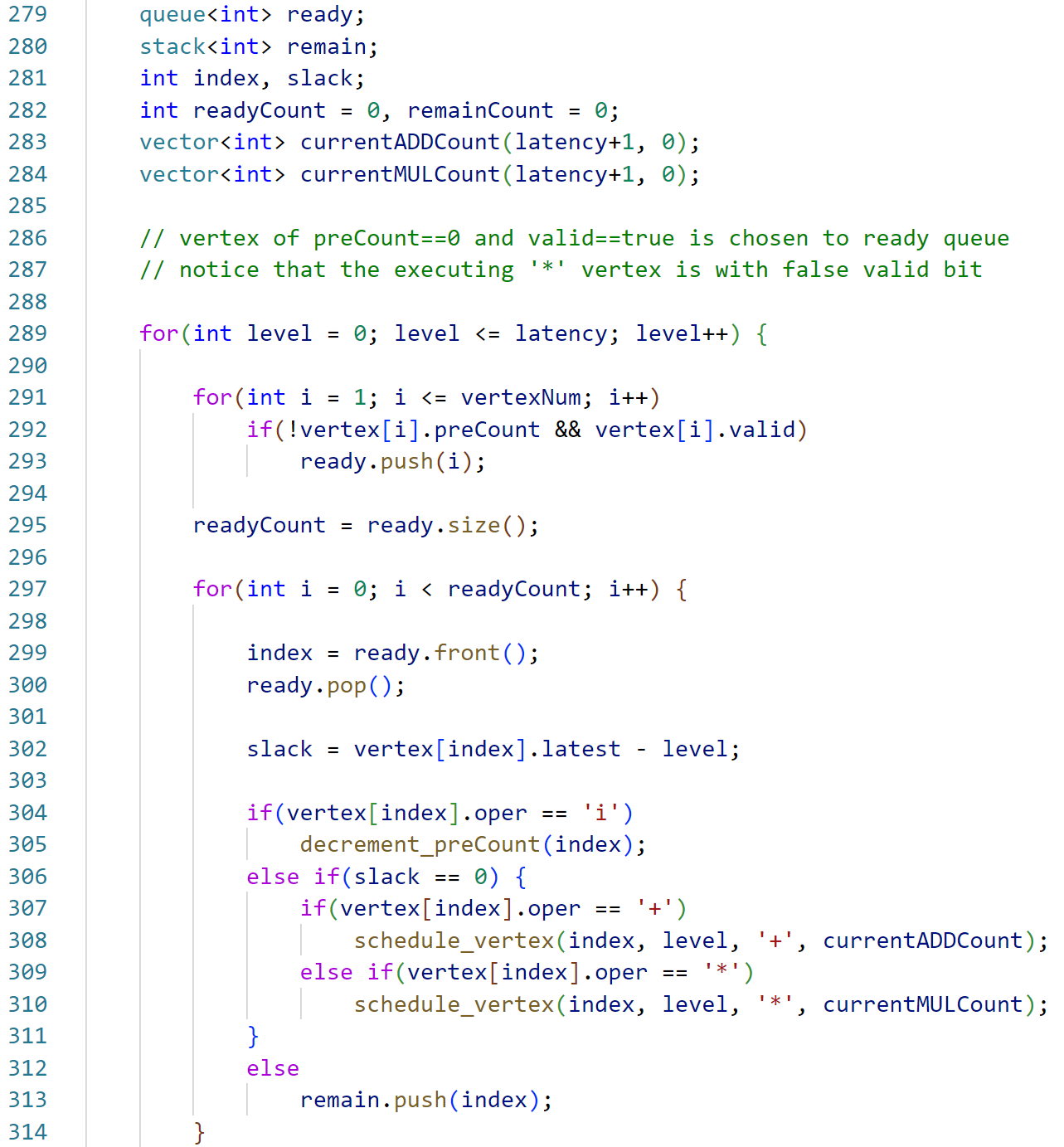


Fig. 18

The for loop (code 320) is to select those vertexes originally in “ready queue”, but now is in “remain stack” because of its nonzero slack. Then, it does the same scheduling procedure as same as the above, only if there are spare hardware left unused (code 325-328).

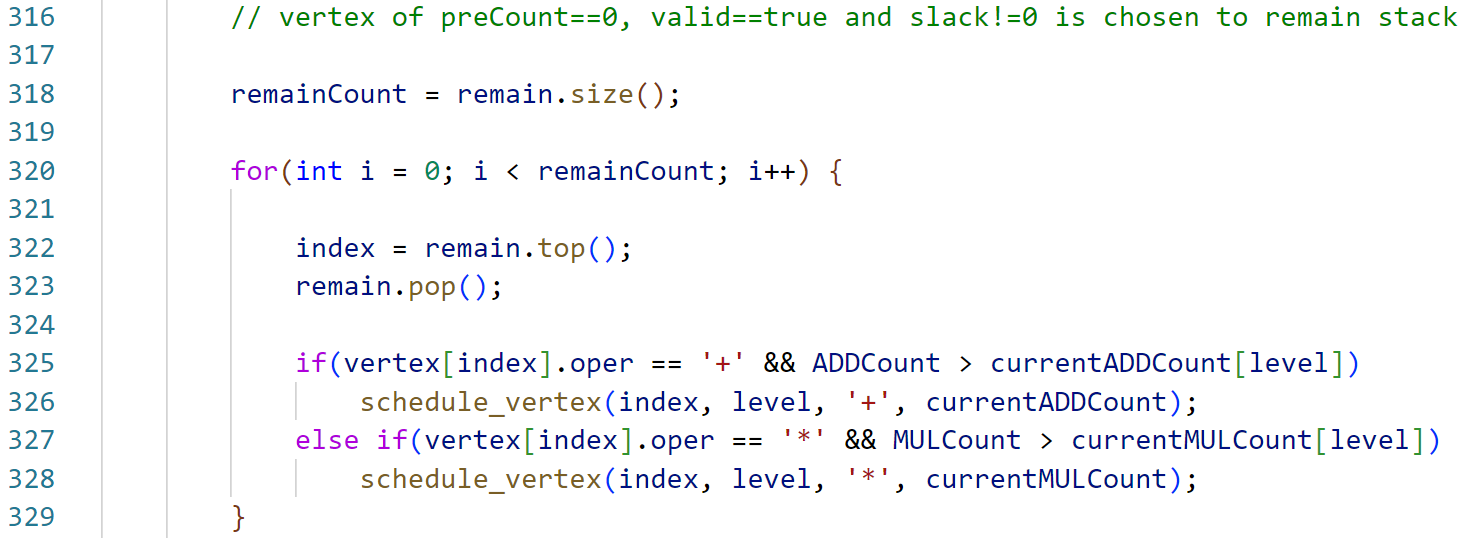


Fig. 19

Then, the for loop (code 332) examines all executing ‘\*’ vertex. It first decreases cycles by 1, and then check if it is equal to 0. If it is 0, then it call *decrement\_preCount()* (to decrease all successor’s *preCount*) (code 333-337) so that this vertex will not include in new MULstate vector. If it is not 0, just add to new MULstate.

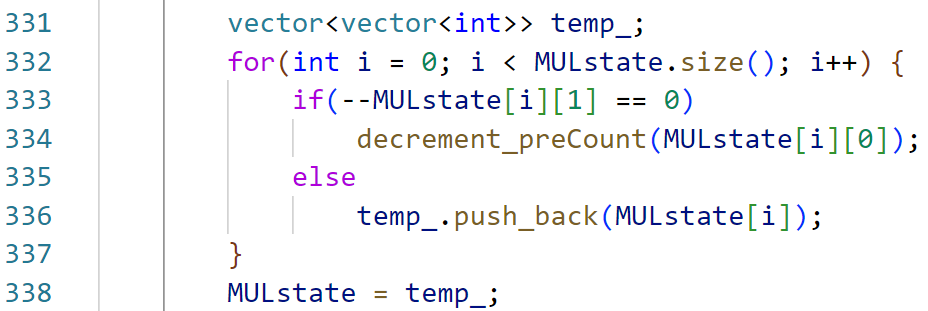


Fig. 20

Afterward, if the current used hardware is more than original one, update it.

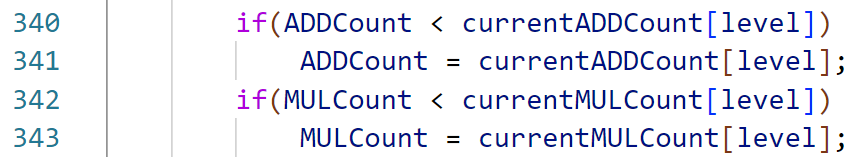


Fig. 21

1. void schedule\_vertex(int index, int level, char oper, vector<int> currentCount)

If it’s ‘+’ vertex, then it is added to output vector. Also, the current adder count will be increased by 1, and it call *decrement\_preCount()* (to decrease all successor’s *preCount*) (code 349-353).

If it’s ‘\*’ operation, then it is added to output vector for 3 cycle (3 space of output vector). Also, the current adder count and the adder count within 3 cylcle will be increased by 1(code 355-359). Then, it adds the index of vertex and cycle number in to MULstate vector (code 360-363). Notice that the *decrement\_preCount()* has not yet been called here, because it’s not the end cycle of operation.

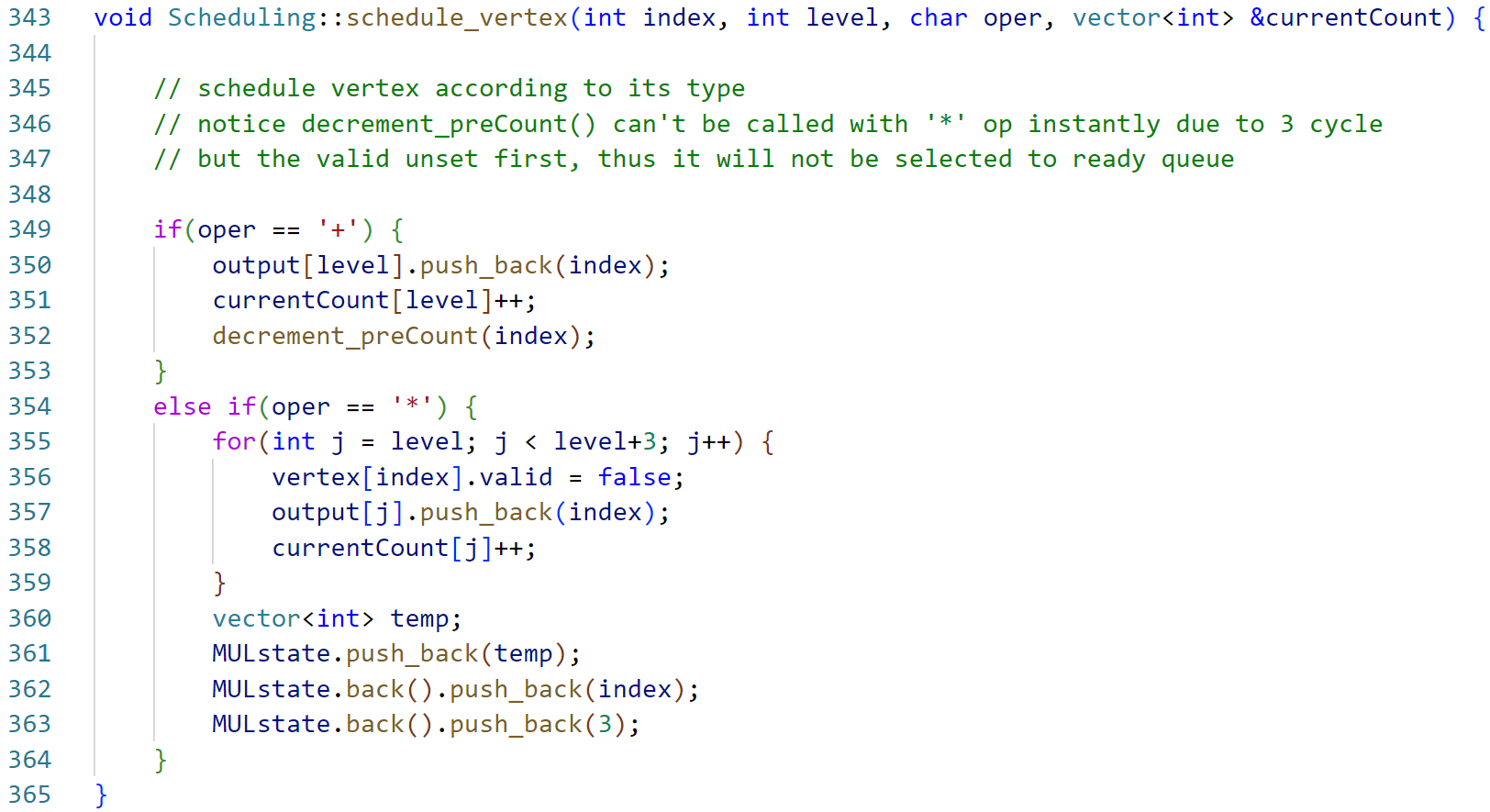


Fig. 22

1. drecement\_preCount(int index)

It is used to decrease all successor’s *preCount* by traversing from headnode.

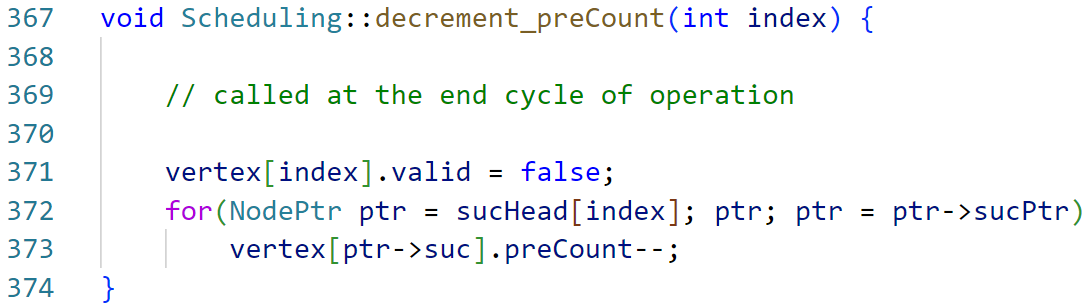


Fig. 23

**Makefile**

Because this project use single .cpp file, there is only one executable file created, i.e., *108501023\_PA2.o*. The following is source code of Makefile.

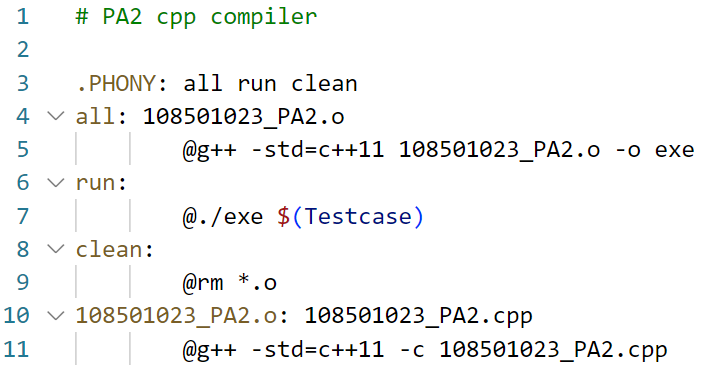


Fig. 24

**Hardness**

The hardest part is the lack of corresponding knowledge while receiving the PA2 document, which, however, is a great chance for me to learn myself. I do some research on google with the relate material, such as the reference from the PA2 document and the chapter 3 of class slide. In this project, I have to review the data structure course, since the data structure I used comes from the chapter 6 including the topology sort concepts and implementations. I am thankful that I have taken the data structure course helping me to come up with useful architecture in mind.

Moreover, I derive some rules with compiler on workstation. Last time, I found that *getline()* would take ‘/n’ as a last input character with the .txt input file in PA1 while it would not doing so with the non-file-extension input file.

**Suggestion**

I am grateful for having this project, it helps me integrating data structure background into this project.