THE GRAPH

DESIGN A SCHEDULE FOR A STUDENT TO COMPLETE HER DEGREE GIVEN THE LIST OF COURSES ALONG WITH THE PREREQUISITES FOR EACH COURSE

STUDENT SCHEDULE

WE HAVE TWO LISTS

LIST OF COURSES

LIST OF PRE-REQS FOR EACH COURSE

THIS CONTAINS PAIRS (COURSE A, COURSE B) WHERE
COURSE A, COURSE B BELONG TO COURSES LIST AND
COURSE A SHOULD BE TAKEN BEFORE COURSE B

WE WANT TO KNOW A VALID ORDER IN WHICH THE STUDENT CAN TAKE HER COURSES!

STUDENT SCHEDULE

WE HAVE TWO LISTS

LIST OF COURSES

CS COURSES
CS 101
CS 102
CS 103
CS 104
CS 105
CS 106
CS 107

LIST OF PRE-REQS FOR EACH COURSE

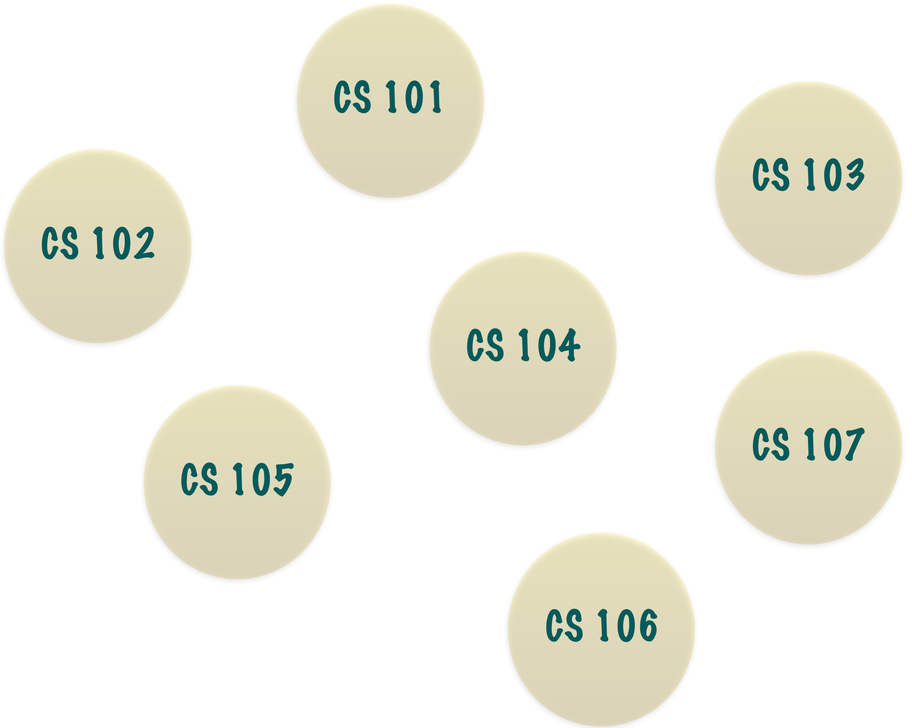
COURSE A	COURSE B
CS 101	CS 102
CS 101	CS 103
CS 103	CS 105
CS 104	CS 105
CS 105	CS 107

STUDENT SCHEDULE

THIS CAN BE MODELED AS A GRAPH PROBLEM

CS COURSES
CS 101
CS 102
CS 103
CS 104
CS 105
CS 106
CS 107

EACH COURSE CAN BE A VERTEX



STUDENT SCHEDULE

THIS CAN BE MODELED AS A GRAPH PROBLEM

COURSE A	COURSE B
CS 101	CS 102
CS 101	CS 103
CS 103	CS 105
CS 104	CS 105
CS 105	CS 107

PRE-REQS ARE EDGES FROM ONE COURSE TO ANOTHER - IT IS A RELATIONSHIP

CS 101 IS A PRECURSOR TO CS 102

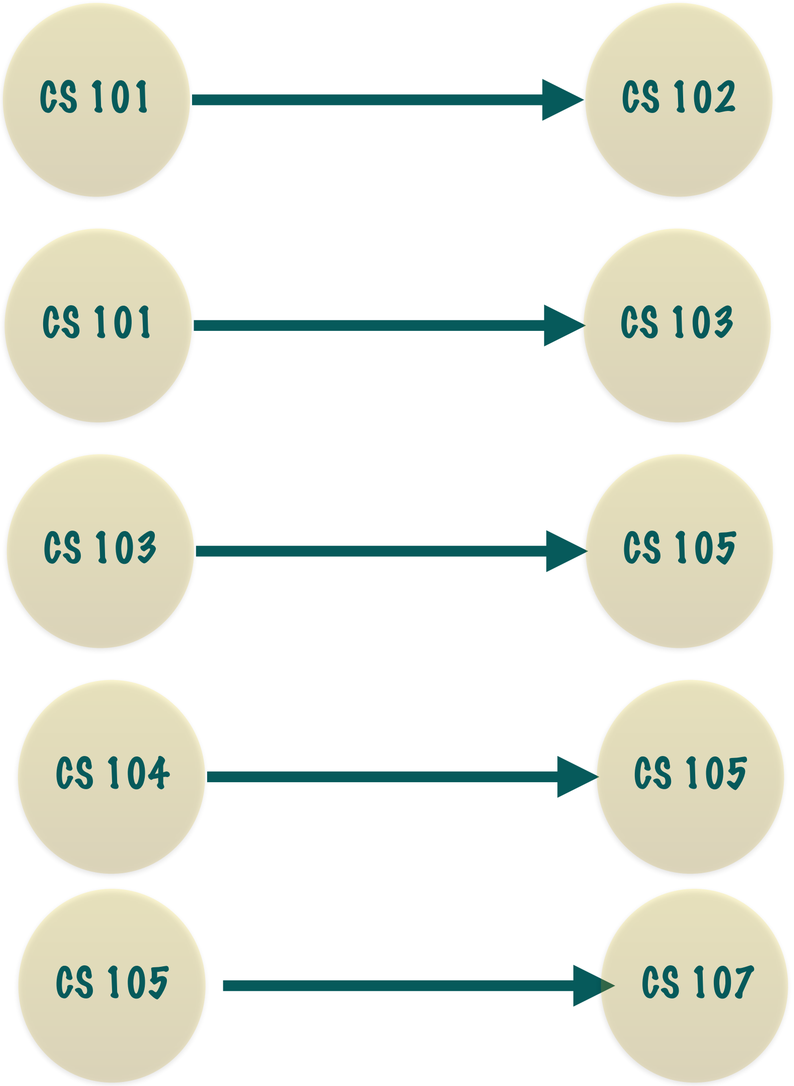
YOU GO FROM CS 101
(COMPLETE CS 101) TO CS 102.

STUDENT SCHEDULE

THIS CAN BE MODELED AS A GRAPH PROBLEM

PRE-REQS ARE EDGES FROM ONE COURSE TO ANOTHER - IT IS A RELATIONSHIP

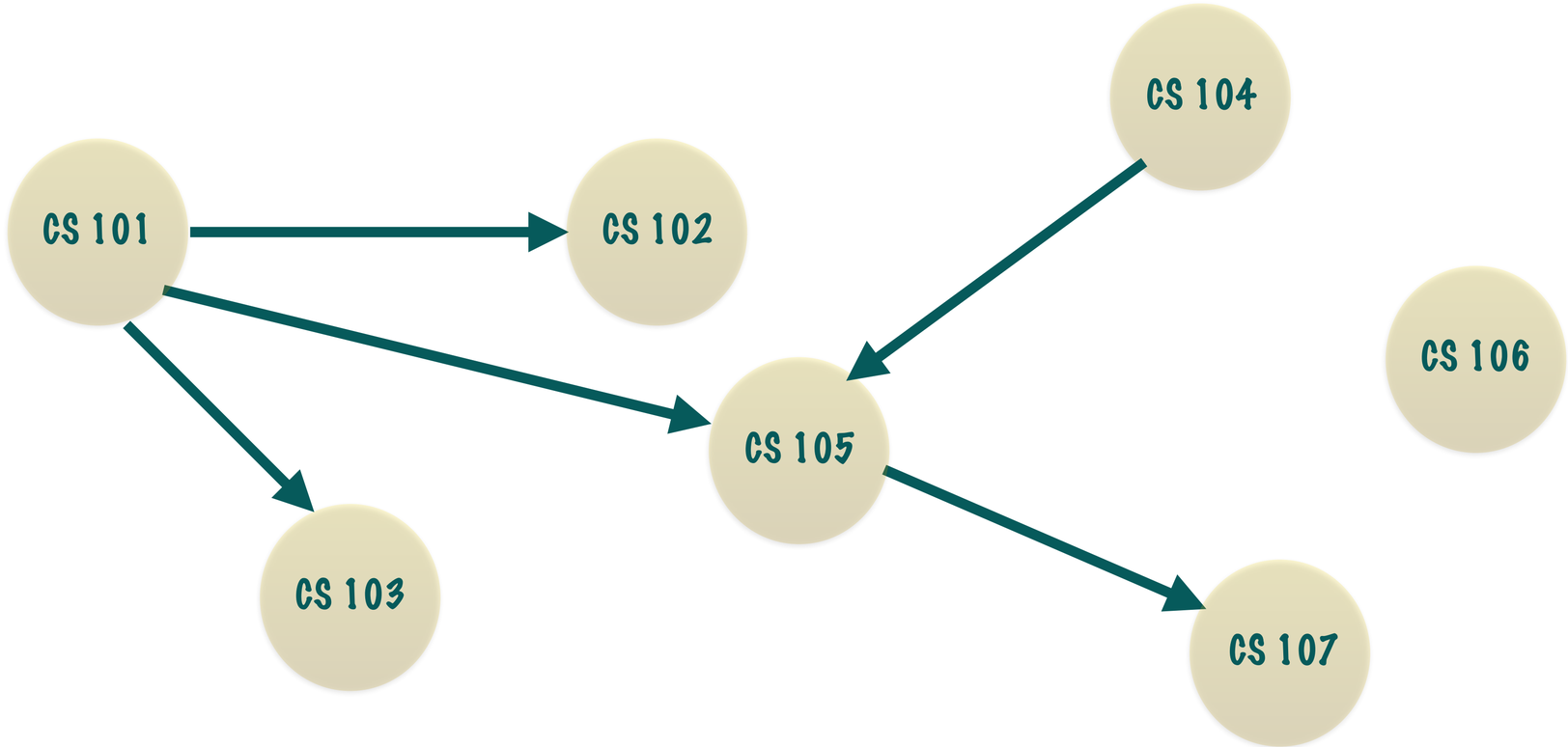
COURSE A	COURSE B
CS 101	CS 102
CS 101	CS 103
CS 103	CS 105
CS 104	CS 105
CS 105	CS 107



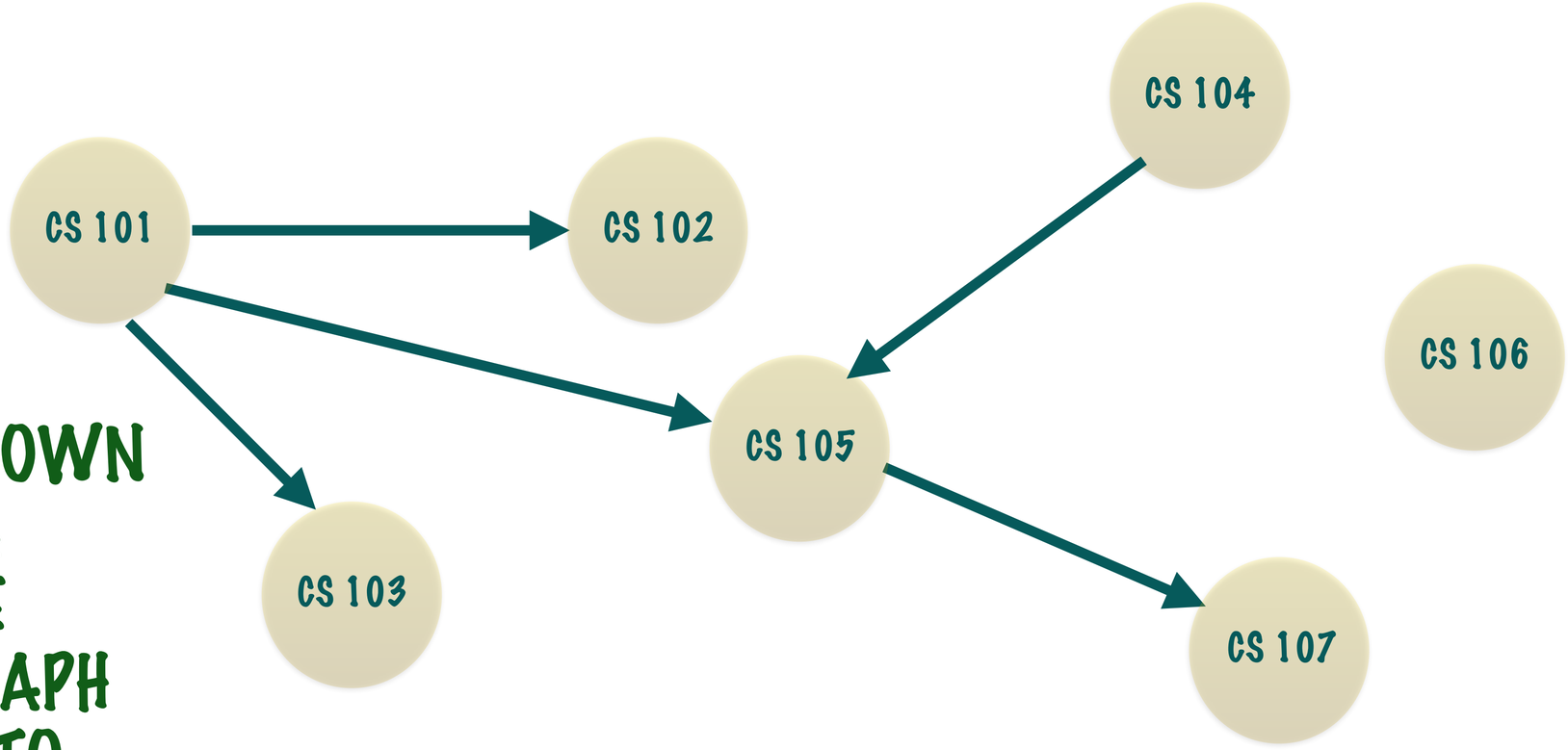
STUDENT SCHEDULE

THIS CAN BE MODELED AS A GRAPH PROBLEM

YOU CAN CONSTRUCT A GRAPH NOW!



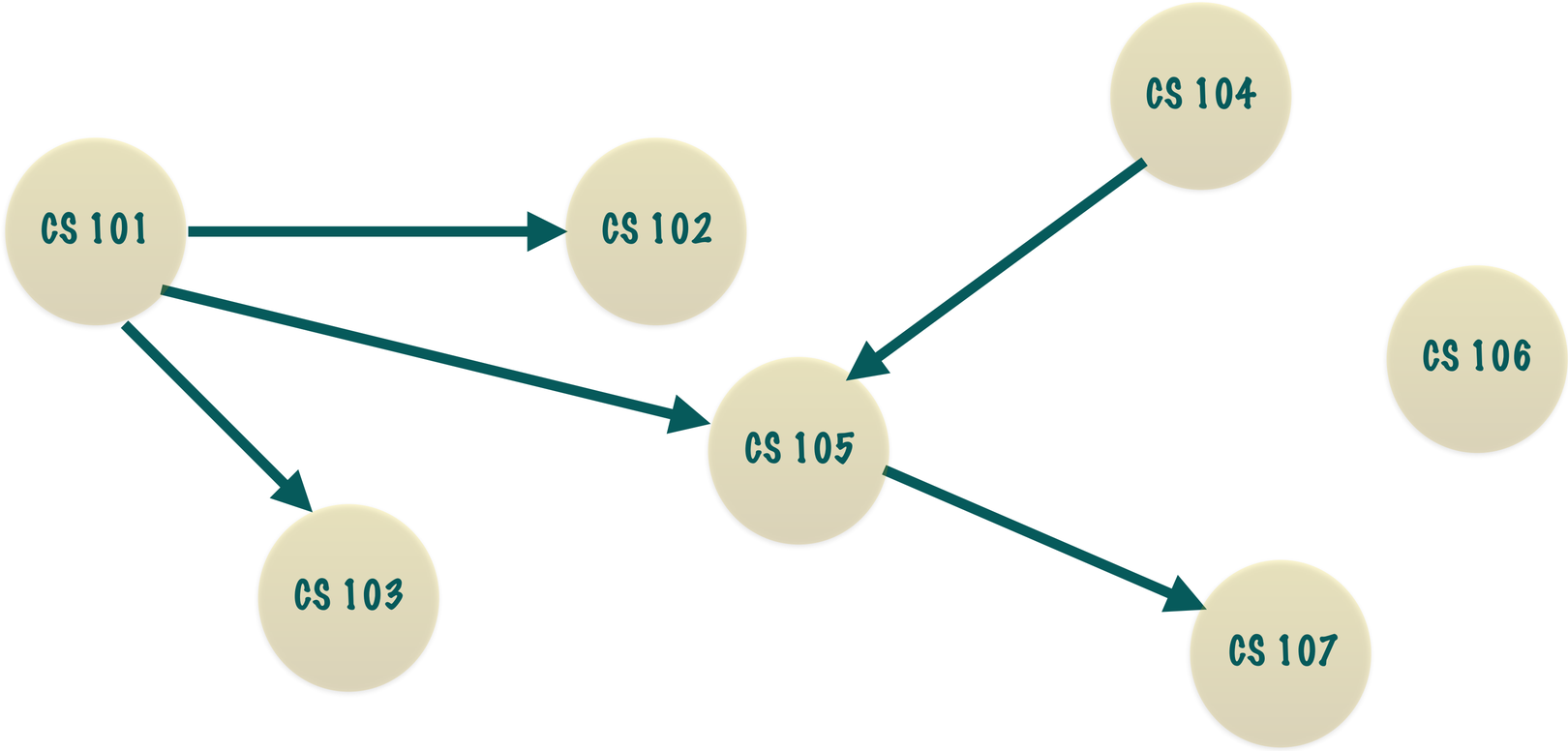
STUDENT SCHEDULE



ONCE YOU BREAK DOWN THE PROBLEM TO A GRAPH SIMPLY USE YOUR FAVORITE GRAPH IMPLEMENTATION TO REPRESENT IT

BUILD A DIRECTED GRAPH WITH UN-WEIGHTED EDGES REPRESENTING THE COURSES AND THE PRE-REQS

STUDENT SCHEDULE



USE THE GRAPH TO FIGURE OUT AN ORDER IN WHICH THE STUDENT CAN TAKE THE COURSES

NOTE - PRE-REQS HAVE TO BE TAKEN BEFORE THE COURSES!

STUDENT SCHEDULE

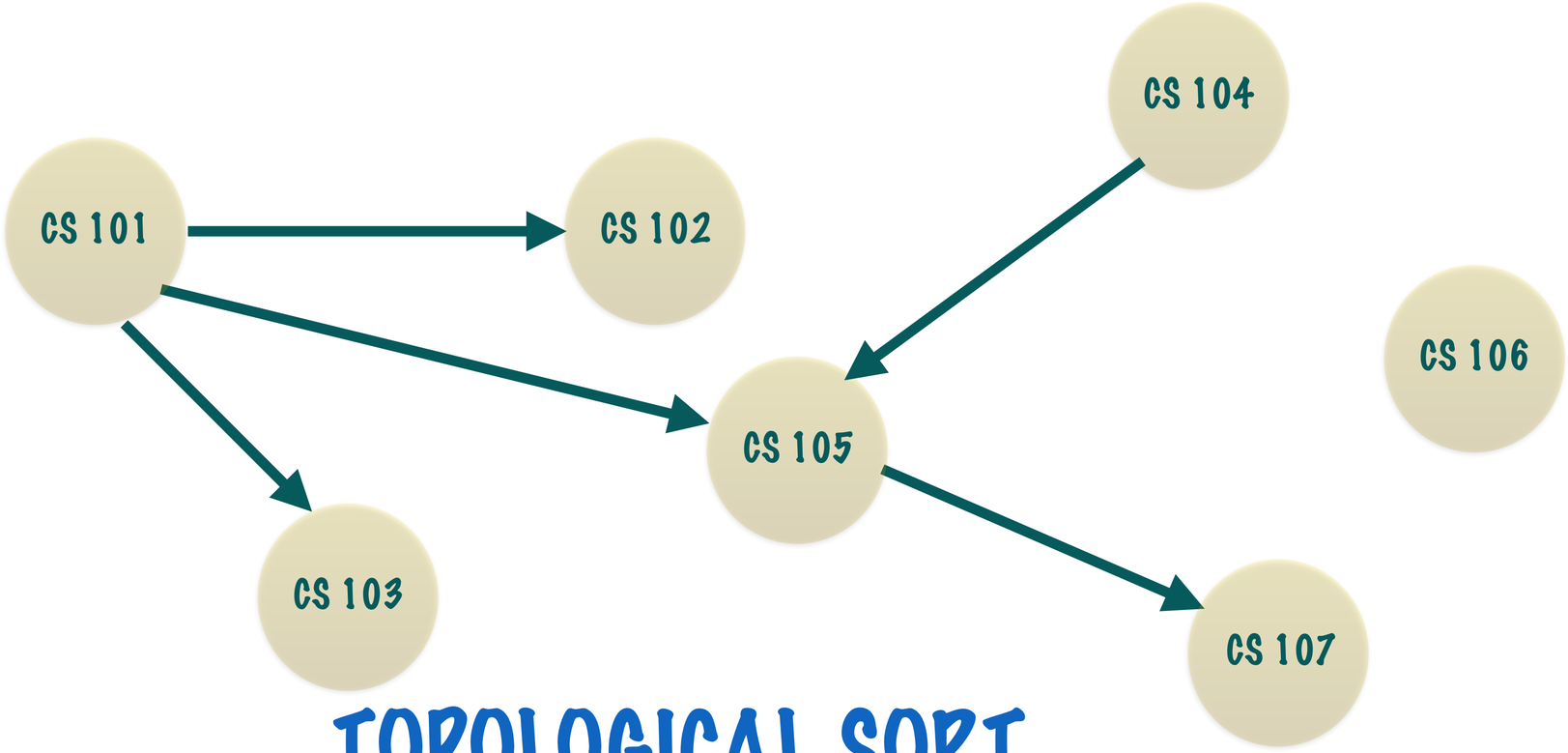
USE THE GRAPH TO FIGURE OUT AN ORDER IN WHICH THE STUDENT CAN TAKE THE COURSES

REMEMBER "TOPOLOGICAL SORT"?

ANY COURSE THAT HAS PRE-REQS SHOULD NOT COME BEFORE ITS PRE-REQS IN THE SCHEDULE!

IT IS AN ORDERING OF VERTICES IN A DIRECTED ACYCLIC GRAPH IN WHICH EACH NODE COMES BEFORE ALL THE NODES TO WHICH IT HAS OUTGOING EDGES

STUDENT SCHEDULE

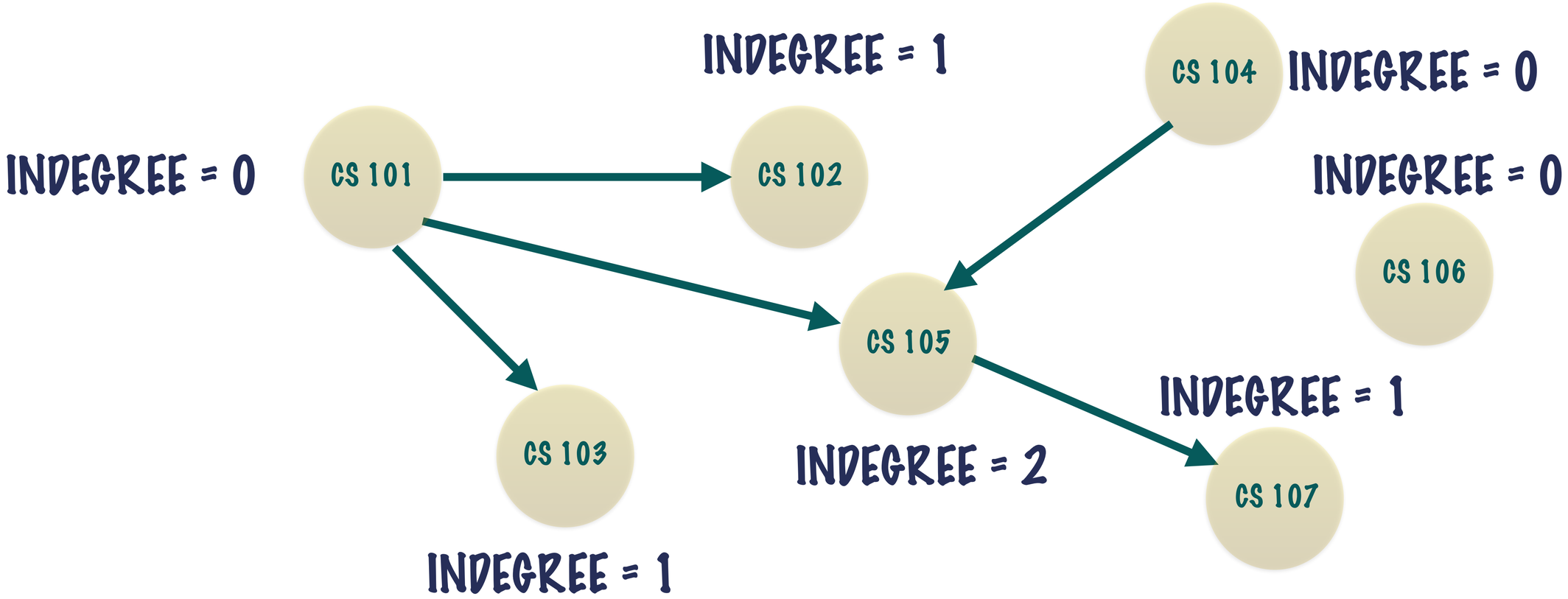


TOPOLOGICAL SORT

IT IS AN ORDERING OF VERTICES IN A DIRECTED ACYCLIC GRAPH IN WHICH EACH NODE COMES BEFORE ALL THE NODES TO WHICH IT HAS OUTGOING EDGES

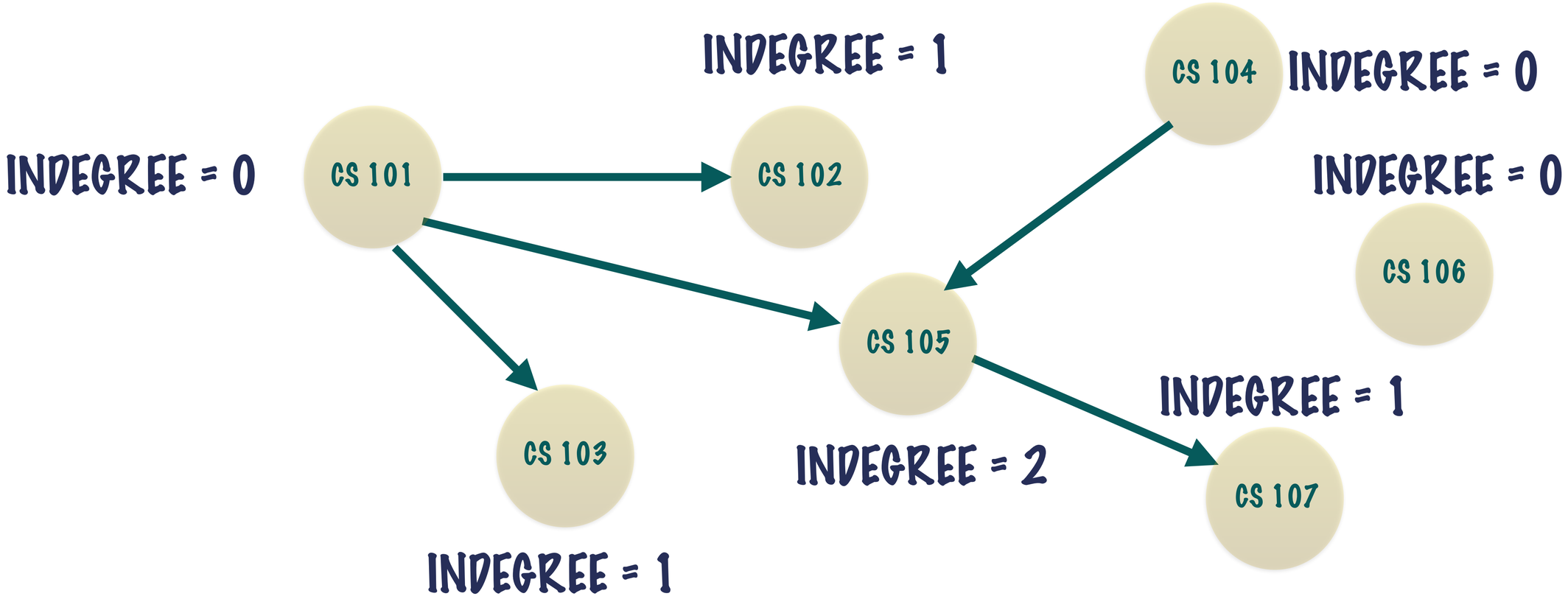
WE CALL OUR TOPOLOGICAL SORT FUNCTION ON THIS GRAPH!

STUDENT SCHEDULE



ALL NODES WITH INDEGREE = 0 ARE POTENTIAL COURSES THE STUDENT COULD START WITH

STUDENT SCHEDULE



THERE ARE MANY SCHEDULES POSSIBLE!

COURSE SCHEDULE

```
public static List<String> order(List<String> courseList, Map<String, List<String>> prereqs) {
    Graph courseGraph = new AdjacencyMatrixGraph(courseList.size(), Graph.GraphType.DIRECTED);

    Map<String, Integer> courseIdMap = new HashMap<>();
    Map<Integer, String> idCourseMap = new HashMap<>();

    for(int i = 0; i < courseList.size(); i++) {
        courseIdMap.put(courseList.get(i), i);
        idCourseMap.put(i, courseList.get(i));
    }

    for (Map.Entry<String, List<String>> prereq : prereqs.entrySet()) {
        for (String course : prereq.getValue()) {
            courseGraph.addEdge(courseIdMap.get(prereq.getKey()),
                    courseIdMap.get(course));
        }
    }

    List<Integer> courseIdList =  TopologicalSort.sort(courseGraph);

    List <String> courseScheduleList =  new ArrayList<>();

    for (int courseId : courseIdList) {
        courseScheduleList.add(idCourseMap.get(courseId));
    }

    return courseScheduleList;
}
```

THE LIST OF COURSES AND THE PRE-REQS ARE INPUTS TO THIS METHOD

SET UP A MAPPING FROM THE COURSE NAME TO A UNIQUE INTEGER ID AND THE REVERSE MAPPING AS WELL

ADD A GRAPH EDGE FOR EVERY PRE-REQ TO COURSE

CALL TOPOLOGICAL SORT ON THE GRAPH

FIND THE COURSE NAMES FOR THE VERTICES AND YOU HAVE A COURSE SCHEDULE